Université catholique de Louvain

Faculté des Sciences Appliquées
Département d'ingénierie informatique

Réalisation d'un système de détection d'intrusion pour Linux, basé sur ASAX

Promoteur : Baudouin Le Charlier
Conseiller : Nicolas Vanderavero
Second lecteur : Xavier Martin

Mémoire présenté en vue
de l'obtention du grade
de diplomé d'études spécialisées
en sciences appliquées
orientation informatique
et de licencié en informatique
orientation informatique générale
par

Christophe Lievens
Corentin Lonfils

Louvain-la-Neuve
Année académique 2003-2004

Remerciements

Nous tenons à remercier toutes les personnes qui ont rendu possible l'aboutissement de ce travail. Nous remercions tout particulièrement Monsieur le Professeur Baudouin Le Charlier et Monsieur Nicolas Vanderavero pour leur disponibilité et les conseils qu'ils ont émis tout au long de sa réalisation. Nous remercions également Monsieur Xavier Martin pour l'aide apportée dans la phase de simulations d'attaques et Madame Bossut Céline pour le temps consacré à la lecture de ce document. Enfin, nous dédions ce travail à Yara et Thomas qui sont venus au monde durant sa réalisation.

Table des matières

Introduction **4**

1 Détection d'intrusions, ASAX, objectifs du mémoire **6**
1.1 Détection d'intrusions . . . 6
1.1.1 Les méthodes de détection . . . 7
1.1.2 Les types d'alertes . . . 7
1.1.3 Problèmes relatifs à un IDS efficace . . . 7
1.2 ASAX . . . 8
1.2.1 Description générale . . . 9
1.2.2 Architecture d'un environnement ASAX . . . 10
1.2.3 Le format NADF . . . 11
1.3 Objectifs du mémoire . . . 12

2 Instrumentation du noyau Linux : concepts et méthodes **14**
2.1 Rappel de concepts théoriques . . . 14
2.1.1 Les librairies . . . 14
2.1.2 Le système d'exploitation . . . 15
2.1.3 Virtual filesystems . . . 20
2.1.4 Démon sous GNU/Linux . . . 21
2.2 Interception des appels système . . . 22
2.2.1 Interposition de librairie . . . 22
2.2.2 Mécanisme de debugging . . . 23
2.2.3 Kernel Patching . . . 26
2.2.4 Loadable Kernel Module . . . 26
2.3 Vers le noyau 2.6 . . . 29

3 Description de divers systèmes existants pertinents pour les objectifs du mémoire **31**
3.1 Les standards C2 et Common criteria . . . 31
3.2 BSM . . . 33
3.2.1 Présentation succincte . . . 33
3.2.2 Architecture . . . 33
3.2.3 Utilité dans le cadre de ce travail . . . 34
3.3 Syscalltrack . . . 35
3.3.1 Présentation succincte . . . 35
3.3.2 Architecture . . . 35

3.3.3 Utilité dans le cadre de ce travail . . . 36
3.3.4 Eléments manquants dans le cadre de ce travail . . . 37
3.4 SNARE . . . 38
3.4.1 Présentation succincte . . . 38
3.4.2 Architecture . . . 38
3.4.3 Utilité dans le cadre de ce travail . . . 39
3.4.4 Eléments manquants dans le cadre de ce travail . . . 39

4 Implémentation d'un mécanisme d'audit pour GNU/Linux, adapté à ASAX 40
4.1 Implémentation du module . . . 40
4.1.1 Mécanisme de hijacking . . . 40
4.1.2 Les fonctions stub . . . 48
4.1.3 Points relatifs à la non-régression du noyau . . . 50
4.1.4 Communication avec le démon . . . 54
4.1.5 Quelques aspects de sécurité relatifs au module . . . 55
4.1.6 Possibilités d'extension . . . 56
4.2 Implémentation d'un démon . . . 56
4.2.1 Introduction . . . 56
4.2.2 Implémentation du démon . . . 57
4.2.3 Réalisation d'un premier client . . . 58
4.2.4 Réalisation d'un client d'analyse temps réel . . . 58
4.2.5 Implémentation d'un mécanisme de configuration . . . 59
4.3 Efficacité de l'implémentation . . . 62
4.3.1 Tests des performances de la machine hôte . . . 62
4.3.2 Simulation de quelques attaques . . . 69
4.4 Comparaison avec les logiciels précédemment analysés . . . 74
4.4.1 Syscalltrack . . . 74
4.4.2 SNARE . . . 74

Conclusion et travaux futurs 77

Bibliographie 79

A Liste des informations auditées : 80
A.1 Process . . . 80
A.2 Appels système . . . 81

B Structure des sources de l'application 82
B.1 Module . . . 82
B.1.1 Répertoire hijack . . . 82
B.1.2 Répertoire perl . . . 82
B.2 démon . . . 82
B.2.1 répertoire nadfd . . . 82
B.3 Le client . . . 83

C Mode d'emploi de l'application **84**
C.1 Déploiement et utilisation 84
C.2 Comment ajouter de nouvelles informations dans les records NADF 85
C.2.1 Informations communes à tous les appels système 85
C.2.2 Informations propres à certains appels système 85
C.2.3 Au niveau du démon. 85

D Template des fonctions stub **86**

E syscall_hijack_private.h **91**

F syscall_hijack_public.h **95**

G unixbench **96**
G.1 system call overhead 96
G.2 spawn 97
G.3 execl 99
G.4 shell 101
G.4.1 looper.c 101
G.4.2 tst.sh 103
G.4.3 multi.sh 104

Introduction

La sécurité des systèmes informatiques a pris de plus en plus d'importance au cours de ces dernières années. Toutefois, la découverte presque journalière de nouvelles attaques et de virus potentiellement dangereux nous montre qu'il est sans doute nécessaire d'utiliser de nouvelles méthodes pour sécuriser les ordinateurs. Une des méthodes qui est devenue populaire est l'utilisation d'un IDS (Intrusion Detection System) permettant de détecter ces attaques.

Parallèllement à ces notions de sécurité, certaines grandes organisations ayant des contraintes à ce niveau (ex la Défense) imposent que les logiciels utilisés aient reçu certaines certifications : C2, Common Criteria, ... Et c'est en partie à cause de ces dernières que GNU/Linux ne peut être déployé au sein de ces grandes organisations.

Un des aspects que doit fournir un logiciel afin d'obtenir certaines de ces certifications (ex Common Criteria) est un mécanisme de détection d'intrusions au niveau du système (ie : audit du système et enregistrement des événements relevants d'un point de vue sécurité). Si certains logiciels GNU/Linux donnent actuellement une réponse partielle à cette préoccupation, il n'existe pas, à notre connaissance, d'implémentations libres proposant d'une part, la possibilité d'auditer les informations intéressantes au niveau du système et d'autre part, d'analyser d'une façon efficace les traces afin de détecter une éventuelle intrusion. Le fil conducteur de ce travail sera donc la réalisation de ce type de logiciel complet. Comme cette tâche représente une certaine importance, nous avons décidé d'orienter notre recherche de solutions en profitant au maximum des opportunités existantes :

- Utilisation d'un logiciel existant et adéquat sans aucune modification : ASAX.
- Réutilisation et adaptation du code de certaines applications : SNARE et Syscalltrack.
- Inspiration d'un système d'audit connu : BSM.

Cette approche nous permettra d'exploiter rapidement les solutions trouvées par les créateurs de logiciels, éventuellement de les améliorer et finalement d'obtenir une solution complète.

Nous présenterons, dans un premier temps, les différents concepts importants relatifs à la détection d'intrusion, effectuerons un rapide survol des possibilités offertes par l'outil ASAX dans le cadre de la détection d'intrusion ainsi qu'une description précise du format spécifique (NADF) utilisé et terminerons par une définition précise des objectifs que nous nous sommes fixés.

Dans le chapitre suivant, nous poserons les différentes bases théoriques nécessaires à la compréhension et à la réalisation d'un mécanisme de détection d'intrusion propre à GNU/Linux et présenterons les différentes techniques pouvant être implémentées.

Ensuite, nous analyserons objectivement différents produits relatifs à la sécurité et à la détection d'intrusion sur GNU/Linux afin de déterminer les avantages et inconvénients de ceux-ci ainsi que la mesure dans laquelle nous pouvons les réutiliser pour obtenir un logiciel répondant à nos objectifs.

Le quatrième chapitre présentera notre implémentation du mécanisme d'audit en insistant sur les aspects les plus importants ainsi qu'en apportant une attention toute particulière aux possibilités d'extensions offertes. Ce chapitre continuera par la présentation des résultats que nous avons obtenus en testant notre logiciel : nous commencerons par un aspect relatif aux pertes de performances du système audité et terminerons par la réalisation de quelques attaques afin de contrôler que le but principal de notre mémoire est atteint.

Enfin, nous terminerons ce travail en énonçant nos conclusions et en proposant les travaux futurs qui pourraient être réalisés dans la continuité de notre mémoire.

Chapitre 1

Détection d'intrusions, ASAX, objectifs du mémoire

1.1 Détection d'intrusions

Une intrusion est un terme générique regroupant le fait d'essayer d'accéder sans autorisation à un système et le fait d'utiliser à mauvais escient des ressources disponibles sur un système. Le terme "mauvais escient" est à prendre dans son sens le plus large et couvre des activités allant de la copie d'informations sensibles et/ou privées à l'utilisation, par exemple, du mail afin d'effectuer du *spam*. La détection d'intrusion est quant à elle définie comme "la détection d'intrus externes qui utilisent un système informatique sans autorisation et d'intrus internes qui possèdent un accès légitime aux systèmes informatiques mais qui abusent de leurs privilèges"[4][1]

Un IDS (Intrusion Detection System) est un logiciel surveillant les activités d'un système ou d'un réseau de systèmes afin de permettre la détection des intrusions sur ces derniers. Le but est donc de se rendre compte que des activités anormales se sont déroulées ou sont en train de se dérouler mais n'est pas d'empêcher la réalisation de celles-ci. Si nous devions faire une analogie avec la vie de tous les jours, nous pourrions comparer un IDS à un système d'alarme domestique[9] : le fait d'installer un système d'alarme n'empêchera pas un intrus de pénétrer dans la maison mais signalera normalement toutes les tentatives d'intrusions.

Les IDS sont typiquement composés de deux parties distinctes :

- Un premier élément (agent collecteur) qui collecte les informations pertinentes d'un point de vue sécurité (les *records*) et les place dans un *audit trail* qui est une séquence chronologique de *records*.
- Un second élément (analyseur) qui exploite l'*audit trail* et détecte les intrusions.

Ils peuvent être de deux types différents :

- Les *host-based* qui permettent la détection d'intrusions au niveau d'un système.
- Les *network-based* qui permettent la détection d'intrusions au niveau du réseau.

[1] traduction libre

1.1.1 Les méthodes de détection

Analyse de signature

La signature est la suite des événements constituant une intrusion. Pour pouvoir être détectée, l'intrusion doit donc être connue et l'IDS doit avoir à sa disposition cette signature. Lorsque l'IDS est actif, il compare la succession des événements qui lui sont soumis avec toutes les signatures qu'il a à sa disposition afin de voir si la suite d'événements qu'il a rencontrée correspond à une attaque connue.

Détection d'anomalie

Il s'agit de détecter les comportements d'utilisateurs s'écartant d'un comportement défini comme normal. Pour détecter ce type d'intrusion, il convient donc de définir les comportements considérés comme normaux et de configurer l'IDS afin qu'il réagisse lorsqu'un comportement s'écarte de ces derniers.

Vérification d'intégrité

Une somme de contrôles est associée aux fichiers d'un système et la vérification de cette somme avec une somme de contrôles de référence permet de se rendre compte d'une modification non-autorisée des fichiers.

1.1.2 Les types d'alertes

Lorsqu'un IDS analyse un *audit trail*, il est susceptible de générer différents types d'alertes. Les alertes du premier type sont les alertes réelles : une intrusion ou une tentative d'intrusion s'est produite et l'IDS l'a détectée. C'est le comportement souhaité d'un IDS et si celui-ci était parfait et correctement configuré, il ne devrait générer que des alertes de ce type à concurrence de une par tentative d'intrusion. Malheureusement, un IDS est susceptible de générer un autre type d'alertes : les *false positives*. Ces dernières sont lancées par un IDS lors de l'analyse d'un *audit trail* alors qu'aucune intrusion ou tentative d'intrusion n'a été perpétrée lors de l'audit du système. Ces alertes ne sont pas souhaitables car elles nécessitent l'intervention d'un administrateur du système audité afin de contrôler si l'alerte générée correspond à une intrusion réelle ou simplement, par exemple, à l'utilisation de mauvaises règles de détection. Nous pouvons nous convaincre de l'utilité d'éviter ce genre d'alerte lorsque l'on sait que des tests réalisés sur une portion de réseau contenant approximativement 20 machines ont montré que près de 15.2 millions d'alertes étaient générées sur une période de 18 heures (parmi lesquelles uniquement 7 heures pendant les heures de travail) et que l'*audit trail* avait une taille de 2.1GB[2].

1.1.3 Problèmes relatifs à un IDS efficace

Maintenant que nous avons certaines notions relatives aux IDS, nous présentons une liste non exhaustive des problèmes qui doivent être pris en compte lors du développement de ce type de logiciel afin qu'il soit utilisable dans un cadre réel.

Stockage des audit trails

Lorsqu'on décide d'effectuer l'audit de systèmes informatiques, il est important d'être conscient que les *audit trails* générés peuvent très rapidement avoir des tailles importantes. Il faut donc prévoir des espaces disques suffisants pour stocker toutes ces informations et des mécanismes permettant de les compresser et de les traiter aussi rapidement que possible afin d'éviter une saturation des espaces de stockage.

Surcharge du réseau

Dans le cas d'un système d'audit centralisé (i.e. chaque ordinateur audite ses événements mais envoie les *records* à un système central responsable de l'analyse de tous les *audit trails*), il faut également prendre en compte le trafic généré entre chaque station auditée et la station centrale.

Stabilité et fiabilité des éléments collecteurs de l'IDS

Etant donné que l'élément collecteur sera intégré au système audité, il est important que celui-ci soit capable de s'exécuter correctement dans tous les types de situations (lorsque le système audité est fortement sollicité, ...), fournisse les informations qui sont effectivement demandées et surtout, ne provoque pas une baisse trop importante des performances du système, voire son "plantage".

Possibilité de configuration

Pour être utilisable, un IDS doit offrir des possibilités suffisantes de configuration. En effet, tous les utilisateurs ne seront pas forcément intéressés par le même type d'information (que ce soit le type d'événements interceptés ou l'information fournie par événement). Il est donc important d'offrir un mécanisme simple de configuration. Dans le même ordre d'idée, ce sont ces capacités de configuration qui permettront de se concentrer uniquement sur les événements intéressants et par conséquent de réduire la taille de l'*audit trail* et, le cas échéant, de réduire l'utilisation du réseau.

Coûts

Afin d'être exploitable dans un contexte réel, il est important que l'utilisation d'un IDS ne provoque pas une baisse trop importante des performances du système audité : le chiffre que nous avons trouvé lors de nos recherches est qu'une baisse de maximum 5% est acceptable lorsqu'on utilise un IDS[11].

1.2 ASAX

ASAX est l'acronyme de "Advanced Sequential file Analysis product on X/Open systems". Etant donné qu'une partie de notre travail a pour but d'obtenir des informations directement exploitables par ce dernier, il est utile que nous présentions cet outil. Toutefois, nous n'en-

trerons pas dans tous les détails[2] et nous invitons le lecteur intéressé par une description plus complète et plus détaillée à consulter [1, 13] [3].

1.2.1 Description générale

ASAX est un logiciel permettant de réaliser une analyse puissante et efficace de fichiers séquentiels. Par puissante, nous entendons le fait qu'il est possible, par exemple, de réaliser les fonctionnalités suivantes[1] :

- Sélectionner certains *records* satisfaisant à certaines conditions pour ensuite les imprimer, les stocker, ...
- Sélectionner une séquence de *records* (pas nécessairement adjacents) satisfaisant à une condition inter-*records*.
- Générer des alarmes, des messages, certaines statistiques.
- ...

Par efficace, nous entendons le fait qu'il est possible d'effectuer plusieurs analyses en ne réalisant qu'une seule lecture des fichiers séquentiels.

Les caractéristiques principales d'ASAX sont au nombre de 5 :

1. Universalité
2. Puissance
3. Efficacité
4. Portabilité
5. Extensibilité

Universalité

ASAX est universel car il rend possible l'analyse de n'importe quel fichier séquentiel grâce :

- Au fait que tous les fichiers analysés doivent être dans un format spécifique à ASAX : le format NADF (Normalized Audit Data Format).
- Au fait que ce format NADF est suffisamment simple et flexible pour permettre une traduction aisée du format original (ADF) vers le format NADF.
- Au fait que la traduction vers le format NADF est réalisée par un *format adapter* qui est soit un programme indépendant d'ASAX, soit trois routines I/O intégrée dans ASAX.

[2] Nous présenterons précisément le format NADF mais ne définirons pas par exemple la syntaxe à utiliser dans certains fichiers.

[3] Ces documents sont également nos références pour la rédaction de cette partie.

Puissance

Les règles qu'ASAX doit effectuer lors de l'analyse d'un *audit trail* doivent être écrites dans un langage créé spécialement pour l'analyse de fichiers séquentiels : RUSSEL (RUles-baSed Sequential Evaluation Language). Grâce à l'utilisation de RUSSEL, il est possible de créer aussi bien des règles d'analyse simples (e.g. afficher le contenu d'un record) que des règles complexes (e.g. ne commencer l'exécution d'une règle que si certaines conditions ont été rencontrées, générer une alarme, ...).

Efficacité

Ceci est un point essentiel car les *audit trails* sont généralement de taille considérable et il faut donc que leur analyse soit tout particulièrement efficace. Nous pouvons affirmer que celle-ci est efficace dans le cas de l'utilisation d'ASAX en raison des points suivants :

- Grâce à l'utilisation du langage RUSSEL, l'analyse de l'*audit trail* est effectué en une et une seule lecture et les informations relatives aux événements pertinents précédents (i.e. les records déjà analysés) sont "encapsulées" dans les règles actives.
- Les étapes répétitives (e.g. analyser le *record* actuel avec les règles actives, passage au record suivant, ...) sont optimisées.

Portabilité

ASAX peut être exécuté sur toutes les machines et tous les *operating systems* disposant d'un compilateur C.

Extensibilité

Le langage RUSSEL supporte un nombre important de fonctions allant de l'analyse simple d'un record jusqu'à des fonctionnalités de statistique. De plus, il a été créé afin d'offrir une interface simple avec le langage C qui permet d'ajouter facilement de nouveaux types d'opérations.

1.2.2 Architecture d'un environnement ASAX

Afin de pouvoir exécuter ASAX, nous devons disposer, en plus du noyau ASAX qui est composé de l'analyseur et de l'ensemble des routines C prédéfinies, des éléments suivants :

- Les fichiers ADF et/ou NADF : il s'agit donc des fichiers contenant les *audit trails* qui doivent être analysés.
- Le *format adapter :* élément nécessaire si les fichiers ne sont pas au format NADF et qui traduira l'information disponible dans l'*audit trail* vers un format NADF.
- Le *data description file* : fichier donnant des informations sur les données qui seront analysées telles que la sémantique, la représentation dans l'*audit trail* original (fichier ADF), la représentation dans le format NADF, ...
- Le programme d'analyse : ensemble des règles écrites dans le langage RUSSEL qui seront utilisées par l'analyseur durant l'analyse du fichier contenant l'*audit trail*.
- Eventuellement, un ensemble de routines C définissant des opérations utilisées dans le programme d'analyse.

Lorsque nous désirons utiliser ASAX, il faut donc exécuter l'analyseur en lui fournissant comme paramètres certains des éléments que nous venons de citer.

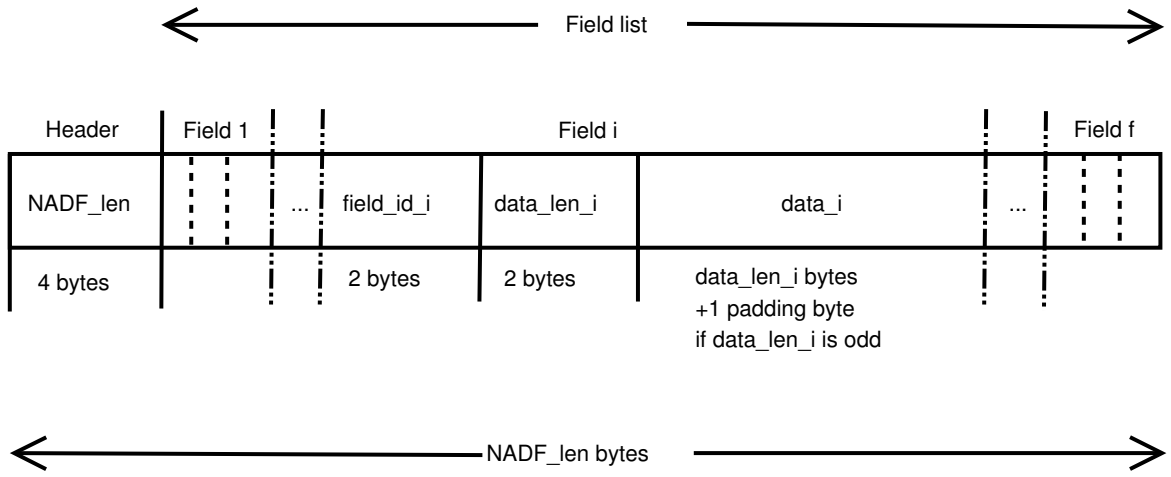

FIG. 1.1 – Un record NADF

1.2.3 Le format NADF

Afin d'effectuer une analyse efficace de l'audit trail, ASAX suppose que les informations qu'il manipule sont au format NADF[1, 13] . Etant donné que le logiciel que nous développons est dédié à ASAX, nous devons donc respecter ce format de sortie. Il est donc utile que nous définissions précisément le format que nous désirons obtenir.

Le respect de ce format est essentiel pour l'utilisation d'ASAX car c'est en partie grâce à ce dernier qu'ASAX effectue le traitement des *audit trails* d'une façon rapide et efficace.

La première caractéristique d'un fichier NADF est le fait que celui-ci commence par un "header record" correspondant à la déclaration C suivante[13] :

```
struct{
     int len; //codé un big endian
     char val[12];
} TypeNADF = {15,"__NADF__1| \0"};
```

Si ASAX ne rencontre pas cette structure en commençant la lecture du flux contenant les informations d'audit, alors le processus s'arrête.

La seconde caractéristique d'un fichier NADF est qu'il contient une succession de records ayant la spécification suivante :

1. Un record NADF (cf figure 1.1) valide est une séquence contiguë de 'NADF_len' bytes (NADF_len >=4) constituée de deux parties distinctes :
   (a) Les 4 premiers bytes du record forment le header dont la valeur correspond à un unsigned integer codé en "big-endian" représentant la valeur NADF_len (ie la taille complète du record diminuée du nombre de bytes de padding ajouté à la fin de celui-ci pour obtenir un alignement sur un multiple de 4 bytes).
   (b) Les bytes suivants forment la séquence des champs contenus dans le record NADF. Chacun de ces champs est à son tour composé des trois parties distinctes suivantes :

i. Les deux premiers bytes (field_id) représentent un "unsigned short" codé en "big endian" dont la valeur est le numéro d'identification du champ. N'importe quelle valeur est possible mais en pratique, les seules valeurs valides sont celles qui sont présentes dans le fichier DDF qui sera utilisé lors de l'analyse du record.

ii. Les deux bytes suivants (data_len) représentent un "unsigned short" codé en "big endian" dont la valeur est la taille en bytes de la dernière partie du champ (data). Si cette valeur est impaire, alors un byte de padding est ajouté à la fin du champ data afin que l'ensemble des trois parties de ce champ soit alignées sur un multiple de 2.

iii. Les data_len bytes suivants représentent l'information contenue dans le record NADF. Si cette information représente une chaîne de caractères, celle-ci est codée de la même manière que la représentation utilisée sur la machine ayant créé le record NADF et si cette information est un integer, alors elle est codée en "big endian". Enfin, si data_len n'est pas un multiple de deux alors 1 byte de padding est ajouté à la suite des data_len bytes (ce byte doit être comptabilisé dans la taille complète du record).

Une condition supplémentaire est le fait que les différents champs du record NADF doivent être triés en ordre ascendant du champ (field_id). Donc si f est le nombre de champs composant le record NADF alors pour chaque champ i tel que $1 <= i < f$, on a field_id_i $<$ field_id_(i+1).

Quelques remarques supplémentaires sur ce format

D'une manière générale, nous remarquons que tous les records (header et normaux) commencent sur un multiple de 4 bytes et que les bytes de padding ajoutés aux différentes fins de records ne sont pas pris en compte dans le champ indiquant la taille complète du record.

Dans le même ordre d'idée, chaque champ d'un record NADF commence sur un multiple de 2 bytes et les bytes de padding ajoutés aux différents champs de données ne sont pas pris en compte dans la taille du champ (mais ils sont bien pris en compte pour la taille complète du record).

Enfin, étant donné que deux bytes de chaque champ sont dédiés à contenir l'information sur la taille de l'information présente (data), il n'est pas utile de mettre le caractère de fin de chaîne de caractère '\0' lorsque l'information fournie est une chaîne de caractères.

1.3 Objectifs du mémoire

Comme le point précédent a permis de le montrer, ASAX constitue une alternative très intéressante pour le composant responsable de l'analyse des *audit trails* d'un IDS complet. Logiquement, le fil rouge de notre mémoire sera donc la réalisation de la première partie de cet IDS : un logiciel permettant d'obtenir les fichiers d'audit.

Avant d'exposer les différents objectifs que nous nous sommes fixés, nous tenons à préciser les aspects qui n'apparaîtront pas dans la suite ce travail :

- Le problème de la surcharge du réseau.
- Le problème du stockage des *audit trails*.

Le premier but, et sans doute le plus académique, est de présenter certaines des possibilités d'implémentation permettant une détection d'intrusion sur GNU/Linux. Nous nous intéresserons uniquement à la détection d'intrusion au niveau du système hôte et ne parlerons pas de la détection d'intrusion au niveau du réseau. Nous n'aborderons également que des possibilités ne nécessitant pas l'instrumentation du code source des programmes s'exécutant sur le système hôte (contrairement au travail de Sergio Norberto Gutiérez Beltran[3]). En effet, le système que nous tentons d'obtenir doit présenter les avantages suivants :

- Pouvoir être déployé sur la majorité des systèmes exécutant une distribution de GNU/Linux sans nécessiter la recompilation des applications.
- Permettre la détection des actions de l'ensemble des processus s'exécutant sur le système (et pas seulement de certaines applications).

Le deuxième objectif que nous nous sommes fixé est de présenter certains logiciels actuels pertinents pour la détection d'intrusion. Cet objectif comporte deux sous-objectifs :

1. Présenter un logiciel de détection d'intrusion complet et connu afin de déterminer les fonctionnalités intéressantes.
2. Présenter des logiciels s'exécutant sous GNU/Linux et ayant certaines particularités intéressantes dans le cadre de la détection d'intrusion. Cette présentation ne doit pas être une énumération des possibilités de chaque logiciel mais bien une synthèse de l'analyse du code source du programme afin d'en dégager les forces, les faiblesses et la mesure dans laquelle ces produits peuvent servir à l'élaboration d'un mécanisme de détection d'intrusion sous GNU/Linux.

Le troisième objectif est de réaliser un logiciel de détection d'intrusion pour GNU/Linux ayant les particularités suivantes :

- Etre dédié à ASAX et donc offrir directement une sortie au format NADF.
- Etre facilement configurable afin de permettre l'audit des informations jugées intéressantes par l'administrateur de la machine.
- Etre facilement modifiable et adaptable afin de pouvoir répondre rapidement à des besoins futurs.

Enfin, notre quatrième objectif est de démontrer la faisabilité d'une solution "complète" endéans les délais impartis pour la réalisation du mémoire et d'évaluer en terme de performances le coût de l'utilisation de notre implémentation ainsi que son efficacité pour détecter des intrusions.

Chapitre 2

Instrumentation du noyau Linux : concepts et méthodes

L'implémentation que nous proposons de réaliser se basera sur l'interception des appels système effectués par les programmes des utilisateurs. La raison principale de ce choix est "basée sur l'observation suivante concernant les attaques : peu importe la nature d'une attaque, les dommages ne peuvent finalement être générés que par des appels système effectués par des processus s'exécutant sur le système cible. Il est donc possible d'identifier (prévenir) les dommages si nous pouvons intercepter chaque appel système effectué par tous les processus et lancer les actions pour prévenir les dommages en annulant l'appel système, changeant ses paramètres ou même en terminant le processus"[8][1]

Afin d'identifier correctement le problème de cette interception, nous commençons par effectuer les rappels nécessaires des notions relatives aux *operating system* en général et à GNU/Linux en particulier. Nous parlons principalement du fonctionnement des appels système sous GNU/Linux et des différentes structures disponibles au sein du noyau. Nous proposons ensuite différentes alternatives permettant d'intercepter les appels système en insistant sur les avantages et inconvénients des différentes techniques. Enfin, nous analysons les implications de la migration d'un noyau 2.4[2] vers le futur noyau 2.6 dans le cadre de l'utilisation des différentes alternatives.

2.1 Rappel de concepts théoriques

2.1.1 Les librairies

Une librairie est un ensemble de variables et de fonctions qui sont compilées et liées ensemble. Elles peuvent être de deux types :

- Statique : lorsque la librairie est liée avec une application au moment de la compilation de cette application. Dans ce cas, une modification du comportement de la librairie après la compilation de l'application n'aura aucun effet sur l'application (elle utilisera toujours la version disponible au moment de la compilation).
- Dynamique : lorsque la librairie est liée avec une application au moment de l'exécution de celle-ci. Ici, une modification du comportement de la librairie après la compilation de

[1] traduction libre

[2] au moment de la rédaction de ce travail, le dernier noyau stable est le 2.4.22

l'application aura des répercussions sur le déroulement de l'application (elle utilisera la dernière version disponible au moment de l'exécution).

Le but de ces librairies est d'éviter de devoir réimplémenter toute une série de fonctionnalités à chaque fois qu'un programmeur crée un nouveau logiciel. Ces dernières sont donc implémentées dans les librairies et mises à la disposition du programmeur qui peut alors se concentrer sur les points les plus importants de son programme.

2.1.2 Le système d'exploitation

Un ordinateur [19] est généralement composé de différents éléments : un ou plusieurs processeur(s), une mémoire principale, des disques durs, un écran, un clavier, des périphériques d'entrées/sorties, une carte réseau, ...

Un des éléments le plus important constituant un ordinateur est le processeur (CPU). Son rôle est d'extraire des instructions de la mémoire et de les exécuter. Il utilise pour cela des registres pour stocker des valeurs intermédiaires de calcul ou des variables. Chaque processeur possède un ensemble d'instructions possibles telles que charger un mot depuis la mémoire dans un registre ou de stocker le contenu d'un registre dans le mémoire. Un processeur a généralement deux modes de fonctionnement : le mode utilisateur et le mode noyau. En mode noyau, le processeur peut exécuter n'importe quelle instruction alors qu'en mode utilisateur les instructions exécutables sont limitées.

Le système d'exploitation (OS) est l'ensemble des programmes qui permettent de gérer les éléments constituant un ordinateur et de fournir aux programmes utilisateurs une interface simple d'utilisation. Il permet de simplifier la vie des programmeurs en cachant la complexité de la prise en charge des différents composants. Un système d'exploitation introduit également un ensemble de concepts de base tels que les utilisateurs, les processus, la mémoire, les fichiers, ... L' OS fontionne en mode noyau, il a donc accès à l'ensemble des instructions du CPU.

GNU/Linux, comme les autres UNIX, est un système multi-utilisateur : il est capable d'exécuter plusieurs applications de manière concurrente (en même temps) et indépendante (sans interaction avec les autres utilisateurs ou les autres programmes). Un utilisateur est une personne possédant certains privilèges sur un système et est identifié par un numéro d'utilisateur appelé UID (*User ID number*). Un utilisateur appartient également à un ou plusieurs groupes qui sont chacuns définis par un numéro de groupe appelé GID (*Group ID* number*)*. Ces groupes sont utilisés par exemple pour réduire l'utilisation de certaines ressources à un nombre restreint d'utilisateurs (ie les membres du groupe).

Il faut encore distinguer un utilisateur particulier sur les systèmes d'exploitation UNIX que l'on appelle généralement *root* ou *super user* et qui possède tous les droits sur le système et peut donc effectuer toutes les tâches sans restriction.

Les processus

Un processus [15] est un concept clé dans le cadre des systèmes d'exploitation et peut être défini comme "une instance de programme en cours d'exécution". Chaque processus est associé avec son *address space* qui est l'ensemble des espaces mémoire pouvant être accédés par celui-ci. Dans cet *address space*, nous retrouvons principalement le programme exécutable, les données du programme ainsi que la pile [19] . Lorsque nous parlons de systèmes d'exploitation multi-tâches, il s'agit simplement de systèmes d'exploitation capables de supporter l'exécution simultanée de plusieurs processus. Dans le cas des machines uniprocesseurs, nous avons l'illu-

sion que l'ensemble des processus s'exécutent en même temps alors qu'il n'y a jamais qu'un et un seul processus actif à un moment donné (pseudo-parallélisme). Les différentes ressources disponibles sur la machine sont donc partagées entre les différents processus s'exécutant et c'est au système d'exploitation d'assurer une distribution optimale de celles-ci. Il faut donc que le système d'exploitation permette à chaque processus de s'exécuter à un certain moment, mais il doit également être capable de maintenir certaines informations relatives à chaque processus (un des exemples les plus intuitifs est sans doute le point de programme atteint par le processus lors de sa dernière activité afin que, lorsque le système d'exploitation lui accorde à nouveau du temps CPU, celui-ci recommence à l'endroit exact auquel il s'était arrêté). Ces informations sont maintenus pour chaque processus présent dans une structure et elles sont toutes stockées dans un tableau appelé *process table*.

Les processus sous GNU/Linux

Si nous analysons le code source de GNU/Linux nous rencontrons souvent le terme ***task*** [15] qui désigne alors un processus. Les concepts que nous venons de voir au point précédent sont appliqués dans le système d'exploitation GNU/Linux. La structure **task_struct** est celle qui maintient l'ensemble des informations relatives à un processus : son état, la liste des fichiers ouverts, l'utilisateur ayant initié le processus, ...

Afin de pouvoir accéder rapidement à l'ensemble des processus s'éxécutant sur le système, le noyau gère également une *process table* qui porte le nom de **task array** et qui contient des pointeurs vers les **task_struct** de tous les processus présents. La taille maximum du **task array** est limitée à **NR_TASKS** processus : ceci signifie donc qu'un système GNU/Linux est capable de gérer simultanément un maximum de **NR_TASK**.

A tout moment, il est également possible d'accéder à la structure **task_struct** du processus s'exécutant effectivement (ie le processus pouvant utiliser le CPU à ce moment) en utilisant la macro **current**. Il est donc possible d'utiliser cette dernière afin de récupérer les informations relatives au processus en cours d'exécution (ex **current->pid** renvoie l'identificateur du processus).

Notion de noyau réentrant

Le terme réentrant [15] signifie que plusieurs processus peuvent être exécutés en mode noyau en même temps et donc qu'une même portion de code du noyau peut avoir plusieurs exécutions concurrentes. Sur un ordinateur uni-processeur, un seul processus peut être exécuté à la fois et les autres processus sont alors bloqués et attendent d'avoir le CPU à leur disposition.

Un exemple permettra d'illustrer l'activité de deux processus dans le noyau : après la lecture sur un disque demandé par un processus, le noyau laisse le périphérique exécuter la lecture et peut alors décider d'exécuter un autre processus en attendant la terminaison de la requête de lecture. Lorsque le périphérique a terminé sa lecture, le processus qui attendait le résultat de la lecture peut continuer son exécution lorsqu'il pourra à nouveau disposer des ressources du CPU.

Une première façon de fournir de la "réentrance" au noyau est d'écrire des fonctions qui ne modifient que des variables locales et n'altèrent pas des variables globales. Ces fonctions sont appelées des fonctions réentrantes. Une seconde manière de permettre la réentrance, qui concerne les fonctions (non-réentrantes) qui accèdent à des variables globales, est d'utiliser des

mécanismes de *locking* afin de s'assurer que l'ensemble des ressources globales soient toujours dans un état consistant : ceci permet d'éviter les problèmes de *race condition.*

Cette notion de noyau réentrant est très importante car sans ce principe, il ne pourrait y avoir qu'un seul processus présent en mode noyau et les performances d'un système seraient diminuées : si nous prenons comme exemple le cas de l'arrivée d'une interruption matérielle, un noyau réentrant sera capable de suspendre le processus courant, même si celui-ci est actuellement en mode noyau, et sera capable de gérer immédiatement l'interruption du périphérique afin de lui permettre de pouvoir effectuer le plus rapidement possible une nouvelle tâche.

Les interruptions

Une interruption [15] est généralement un événement qui altère la séquence d'instructions exécutée par un processeur : lorsque que celui-ci reçoit un signal d'interruption, il arrête la tâche courante pour exécuter la gestion de l'interruption. Le but d'une interruption est donc d'informer le CPU qu'un événement particulier vient de se produire (ex : une opération de lecture s'est terminée) afin qu'il puisse effectuer un certain nombre d'actions relatives à cet événement. Le type de transmission de ces événements correspond souvent à un signal électrique généré par un composant matériel de l'ordinateur. Mais il est également possible pour un logiciel de générer des interruptions : les interruptions logicielles (*software interrupts).* Le noyau Linux n'implémente qu'une seule interruption logiciel qui est l'appel assembleur int $0x80 utilisé lorsqu'un processus demande l'exécution d'un appel système : lorsque cette instruction est exécutée, le CPU bascule du mode utilisateur vers le mode noyau afin de réaliser l'appel système.

Les appels système

La notion d'appel système [15] [19] nous intéresse tout particulièrement car l'implémentation que nous proposons est fortement liée à celle-ci. Un appel système est une interface entre le système d'exploitation et les programmes utilisateurs. Cette interface permet aux programmes utilisateurs d'interagir avec le noyau afin de, par exemple, utiliser une ressource ou créer un nouveau processus.

Les appels système sont donc le moyen à disposition des processus s'exécutant en mode utilisateur d'interagir ou de demander des services au système d'exploitation[19] . Le fait de placer ces niveaux supplémentaires d'abstraction (interface) entre les processus et le système d'exploitation présente de nombreux avantages :

- La programmation des applications est plus aisée car il n'est pas utile de s'inquiéter des particularités de bas-niveau.
- La sécurité est renforcée car le noyau a la possibilité de vérifier les requêtes avant leur exécution.
- L'utilisation des interfaces permet de créer des applications plus portables car indépendantes de l'architecture de la machine : en effet, l'invocation de ces appels système n'est pas réalisée directement par les programmes utilisateurs mais ces derniers utilisent des fonctions disponibles dans des librairies et ce sont celles-ci qui invoquent réellement les appels système.

Afin de bien comprendre les différentes étapes réalisées lors de l'exécution d'un appel système en général, avant de s'intéresser à l'implémentation particulière de GNU/Linux, nous

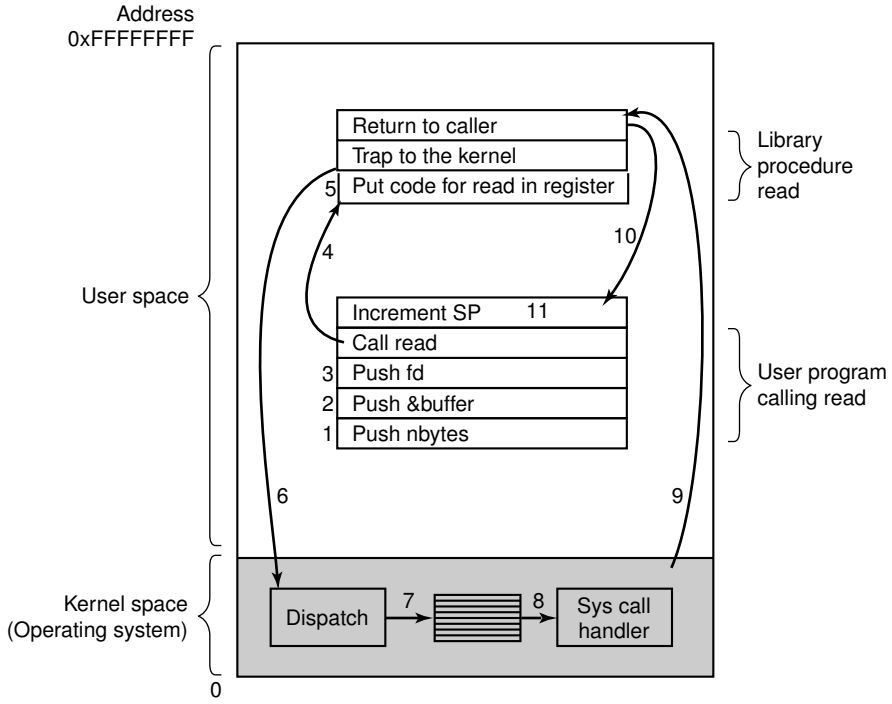

FIG. 2.1 – Les différentes étapes de l'exécution de l'appel système *read*

proposons de parcourir un exemple extrait du livre de Tanenbaum[19] en utilisant la figure 2.1[3] : l'appel système *read*.

L'appel de cet appel système peut s'effectuer de la façon suivante dans un programme C :

```
count=read(fd,buffer,nbytes);
```

L'exécution de cette ligne aura normalement pour effet de placer un entier correspondant au nombre de bytes (généralement égale à *nbytes*) lu dans le fichier référencé par *fd* dans la variable *count* et de copier ces bytes dans *buffer*. Par contre, si un problème survient lors de l'exécution de l'appel système, alors *count* recevra la valeur -1.

L'exécution de cette ligne de code se déroule de la façon suivante :

- Le programme place les différents paramètres sur la pile *:* 1, 2, 3 sur la figure.
- Un appel vers la fonction de la librairie est effectué : 4 sur la figure.
- La fonction invoquée dans la librairie place alors les différents paramètres (disponibles sur la pile) à un endroit bien spécifique où l'OS s'attend à les retrouver (ex les registres) : 5 sur la figure.
- L'instruction **TRAP** est alors réalisée afin de passer de mode utilisateur au mode noyau et une partie de code du noyau (*Dispatch* sur la figure) commence alors à être exécutée afin d'inspecter le numéro de l'appel système qui est invoqué et, en fonction de celui-ci, demander à un certain *handler* d'exécuter et effectivement les instructions de l'appel système : 6, 7 et 8 sur la figure.

[3]Cette figure est disponible sur le site de Tanenbaum à l'url suivant : `http://www.cs.vu.nl/~ast/books/book_software.html`

– Une fois que le *handler* a terminé son exécution, le contrôle est redonné à la fonction de la librairie qui pourra alors exécuter les instructions suivant le **TRAP.** Une fois que ces instructions sont effectuées, c'est le programme utilisateur qui récupère le contrôle et celui-ci n'a plus qu'à retirer de sa pile les éléments qu'il avait placés pour effectuer l'appel système : 9,10 et 11 sur la figure.

Les appels système sous GNU/Linux

A présent que les notions générales relatives aux appels système ont été définies, nous nous attardons un peu plus sur l'implémentation concrète qui est réalisée dans les systèmes GNU/Linux. Afin de faciliter la vie des programmeurs, celui-ci propose donc toute une série de librairies de fonctions que ces derniers peuvent utiliser. Certaines de ces API's (définies dans le "*libc* standard C library") sont des *wrapper routines* dont le seul but est d'effectuer un appel système. Ces *wrapper routines*[4] définissent donc l'API qui doit être utilisée par les programmes pour effectuer des appels système. En règle générale, chaque appel système correspond à une *wrapper routine*. Par contre, une *wrapper routine* ne correspond pas forcément à un appel système : il est possible que la routine offre des services en mode utilisateur ou qu'elle invoque plusieurs appels système durant son exécution.

Lorsqu'un processus demande l'exécution d'un appel système, le CPU passe du mode utilisateur au mode noyau et commence l'exécution d'instructions dans le noyau. Sous GNU/Linux, les appels système sont invoqués en utilisant l'instruction assembleur int $0x80 (*software interrupt)* et en plaçant les différents paramètres permettant l'exécution de l'appel système dans les registres (ex : *system call number*, qui permet d'identifier l'appel système demandé, est placé dans le registre **eax**).

Le *system call handler* est utilisé pour effectuer les opérations communes à tous les appels système : enregistrer le contenu des registres dans le **kernel mode stack** (pile utilisée par le processus lorsqu'il est exécuté en mode noyau) et exécuter l'appel système en invoquant la fonction C correspondante appelée *system call service routine*. Enfin, on quitte le *systemcall call handler* à l'aide la fonction *ret_call_from_sys_call()*.

Si nous mettons cela en parallèle avec l'explication donnée au point précédent, nous obtenons : le *dispatch* (*system call handler)* utilise un numéro (*system call number*) afin de déterminer et d'atteindre le bon *handler* (*system call service routine)*. Ceci est rendu possible sous GNU/Linux en utilisant un tableau (**syscall_table**) contenant des pointeurs vers les *handlers* (*system call service routine*) et le *system call number* qui est passé lors de la demande d'exécution d'un appel système n'est rien d'autre que l'index correspondant à l'appel système dans la **syscall_table.** Ce tableau a une taille de **NR_syscalls** (généralement 256) qui est simplement une macro limitant le nombre d'appels système qu'il est possible d'implémenter. Si un numéro ne correspond à aucun appel système présent sur le système hôte, alors l'entrée de la **sys_call_table** pour ce numéro pointe vers l'adresse de la fonction **sys_ni_syscall()** qui est une routine pour tous les appels système non implémentés.

Dans la figure 2.2, on trouve graphiquement les différentes étapes de l'invocation d'un appel système dans le cas de GNU/Linux.

Les différentes grandes étapes composant l'exécution interne du *system call handler* sont les suivantes :

– Le numéro de l'appel système et les valeurs des registres du CPU susceptibles d'être utilisées sont sauvés.

[4] Il s'agit donc de routines dont au moins une des instructions effectue un appel système.

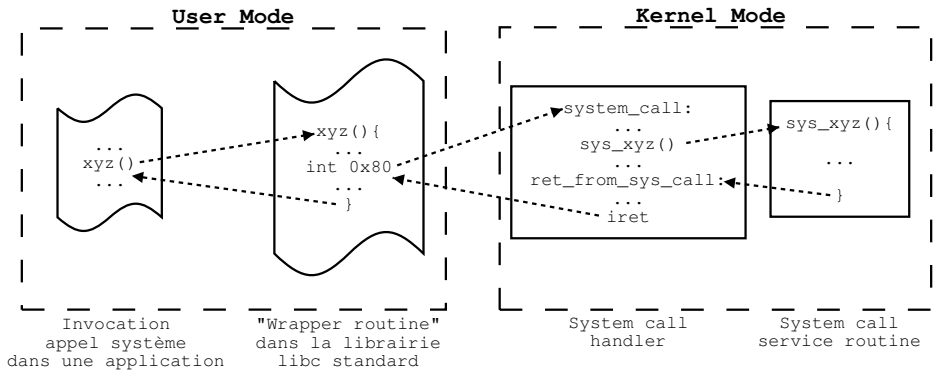

FIG. 2.2 – Invocation d'un appel système.

- Test de validité sur le numéro de l'appel système : s'il est plus grand ou égal à **NR_syscalls** alors le system call handler se termine.
- Vérification si le flag **PF_TRACESYS** inclus dans le champ flags de **current** est égal à 1, ce qui signifie que les appels système effectués par le programme sont requis par un autre programme (ex : dans le cadre de debugging). Le cas échéant, **system_call**() invoque la fonction **syscall_trace**() avant l'exécution de la routine et une seconde fois après l'exécution de la routine. L'effet de **syscall_trace**() est d'arrêter l'exécution de **current** et de permettre à l'autre programme de collecter des informations sur **current**.
- Exécution de la routine spécifique associée avec le numéro de l'appel système : le *system call service routine*.

2.1.3 Virtual filesystems

Le noyau Linux [16] fournit un mécanisme qui donne accès par une arborescente de fichiers aux informations relatives à l'état du système. Cette interface de communication permet également aux utilisateurs d'accéder à certaines structures de données du noyau Linux et de communiquer avec ses composants.

Les **virtual filesystems** sont donc des fichiers spéciaux sur les systèmes d'exploitation GNU/Linux qui sont placés dans le répertoire /proc et qui ne jouent pas de rôle dans la gestion de l'espace des disques classiques : ce sont des fichiers virtuels qui sont maintenus en mémoire et qui n'occupent donc pas d'espace sur les disques physiques. Le fichier /proc/cpuinfo est un exemple de fichier virtuel : lorsque l'on effectue la commande "cat /proc/cpuinfo" depuis une console, nous obtenons un ensemble d'informations relatives au CPU (cf figure2.3). Pour l'utilisateur, il n'y a donc pas de différence entre l'accès à un fichier classique et un fichier virtuel.

Pour pouvoir réaliser ce type de fichier virtuel, il est nécessaire de réaliser un *driver* implémentant les fonctions qui seront exécutées lorsqu'un processus accédera à ce fichier virtuel : pour un processus qui désire lire un fichier en exécutant classiquement un appel à *read*, le *driver* implémentera la fonction **read_proc()** qui placera les données à renvoyer dans le buffer fourni par le processus lors de sa demande de lecture afin que celui-ci puisse y avoir accès lorsque l'exécution du *read* se termine.

```
processor : 0
vendor_id : AuthenticAMD
cpu family : 6
model : 10
model name : mobile AMD Athlon(tm) XP2500+
stepping : 0
cpu MHz : 1855.081
cache size : 512 KB
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 1
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 sep mtrr pge mca cmov pat pse36 mmx fxsr
sse syscall mp mmxext 3dnowext 3dnow
bogomips : 3670.01
```

FIG. 2.3 – Exemple fichier virtuel /proc/cpuinfo

Une seconde fonction intéressante est la fonction *ioctl*. Lorsque le *driver* supporte cette fonction, il est alors possible de l'utiliser pour communiquer et configurer le *driver* "à chaud". Les arguments requis par cette fonction sont :

- Un *file descriptor* correspondant au fichier virtuel.
- Un int représentant la commande qui doit être exécutée.
- Un paramètre optionnel d'une taille de 4 bytes : ceci permet de fournir des informations directement utilisables par le driver ou un pointeur vers une structure plus importante à laquelle le *driver* pourra accéder.

En conclusion, il est donc possible de communiquer dans deux sens avec les éléments du noyau en utilisant ces mécanismes. La fonction *read* nous permet de communiquer du noyau vers l'espace utilisateur et la fonction *ioctl* nous permet de communiquer de l'espace utilisateur vers le noyau. Plus d'informations sur l'implémentation et l'utilisation des *drivers* et des fichiers virtuels sous GNU/Linux sont disponibles dans le livre [16].

2.1.4 Démon sous GNU/Linux

Un démon [10] est un processus qui est exécuté en arrière-plan des applications et qui n'est pas lié à un *shell* particulier. Généralement, ce processus débute quand le système démarre et reste actif durant tout le temps d'activité du système.

L'exécution du démon se fait à partir d'un script placé dans le répertoire /etc/init.d. *Init* est le processus parent de tous les processus. Il crée différents processus sur base du script que l'on trouve dans le fichier /etc/inittab. Chacun des processus créés par *Init* est à l'origine d'autres processus (par exemple le processus de login par lequel les utilisateurs auront la possibilité de se logger).

Le démarrage des démons se fait généralement à l'aide d'un script *bash* demandant un argument pouvant prendre trois valeurs différentes :

- *start* pour démarrer le démon
- *stop* pour arrêter le démon
- *restart* pour redémarrer le démon.

En règle générale, ces scripts sont exécutés directement au démarrage de la machine.

2.2 Interception des appels système

A présent que certains concepts relatifs aux systèmes d'exploitation ont été posés et que le mécanisme général de l'exécution d'un appel système est connu, nous pouvons intuitivement découvrir les différentes façons d'intercepter les appels système (i.e. en plaçant un programme entre les différentes indirections). La suite de ce point présentera donc en détail les aspects de cette interception en fonction des endroits où nous plaçons notre programme. Pour chacun des différents points, nous présenterons la technique utilisée, les avantages et inconvénients de cette technique ainsi qu'une étude du coût[5] engendrée par la mise en place de la technique.

2.2.1 Interposition de librairie

Bien que cette méthode ne soit pas à proprement parler une instrumentation du noyau GNU/Linux, nous avons décidé de la mentionner dans ce chapitre afin d'être complet et également parce que celle-ci présente la caractéristique d'être générale pour l'ensemble des programmes s'exécutant sur une machine.

La méthode d'interposition consiste à exploiter le principe des librairies dynamiques et de placer une nouvelle librairie de fonctions entre l'application et sa référence vers une autre librairie de fonctions. Cette manipulation est rendue possible sous GNU/Linux en ajoutant une variable d'environnement LD_PRELOAD avant l'exécution du programme que nous désirons monitorer. Quand le programme effectuera un appel de fonction non défini, le *dynamic linker* effectuera une recherche de la définition de cette fonction dans les objets présents dans la variable LD_PRELOAD avant d'effectuer une recherche dans le chemin normal des librairies. La figure 2.4 montre l'utilisation de ce mécanisme.

Avantages

- Facilité d'implémentation.

Inconvénients

- Technique "facilement" contournable : il suffit par exemple d'écrire un programme qui invoque les appels système en utilisant un mécanisme de plus bas niveau (e.g. la commande assembleur INT 0x80 sur un GNU/Linux X86).
- Les programmes "root" où le *setuid* est placé ne tiendront pas compte de la variable d'environnement LD_PRELOAD sauf si le UID[6] est celui du root.

[5] Il s'agit du coût en terme de temps d'exécution sur la machine hôte

[6] UID (User IDentifier) : integer identifiant de façon unique un utilisateur

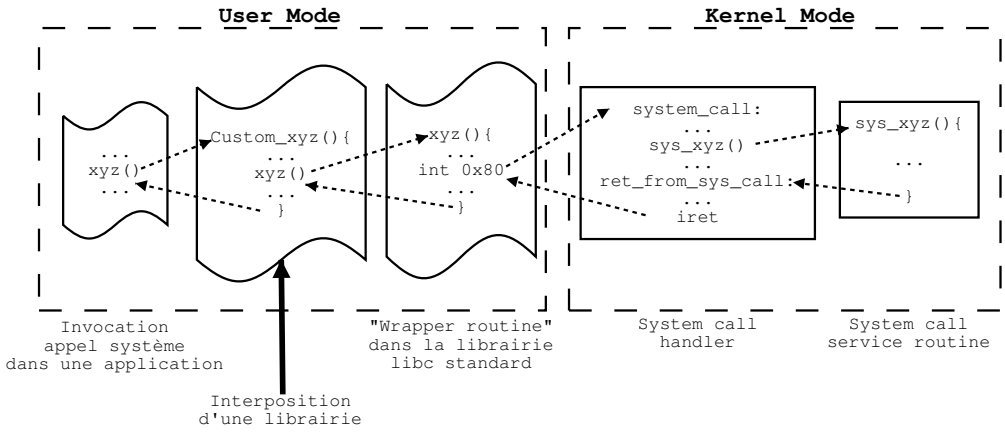

FIG. 2.4 – Interposition de librairie

– Certains appels système effectués lors d'attaque de type "buffer overflow" depuis le code shell n'ont pu être détectés car ces attaques n'utilisent pas les librairies pour effectuer les appels système [17].

Coûts

Aucune étude contenant des informations relatives aux coûts d'une implémentation de ce type n'a été trouvée mais étant donné les inconvénients que cette technique présente, nous n'avons pas jugé nécessaire d'effectuer une recherche plus approfondie.

2.2.2 Mécanisme de debugging

Le système GNU/Linux possède des mécanismes permettant à un processus d'intercepter et d'enregistrer les appels système effectués par un autre processus. La fonction *strace()* permet par exemple de contrôler l'exécution d'un processus enfant. Le but premier de ces mécanismes est une utilisation dans le cadre du debugging d'une application :

– Le processus enfant se comporte normalement jusqu'à ce qu'il soit stoppé par le noyau.
– Le processus parent est alors notifié via la fonction *wait()* et peut alors examiner l'état du processus enfant.
– Une fois que les informations sont collectées, le parent permet la continuation du processus enfant.

Avantages

– Portabilité du code d'extension si on se conforme à une API standard.

Inconvénients

– Ajout de deux context-switch supplémentaires pour chaque appel système des processus monitorés.

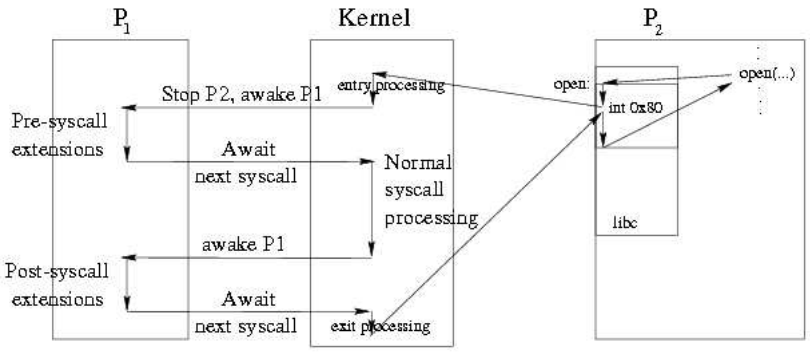

FIG. 2.5 – Exécution d'un appel système pour un processus monitoré

- Le processus effectuant le monitoring se trouve dans le "user space", ce qui signifie qu'il ne peut pas accéder à des informations spécifiques du processus monitoré (e.g. les variables d'environnement).
- L'utilisation d'un mécanisme pur de debugging tel que strace peut provoquer le crash de certains programmes (e.g. sendmail) et ralentir l'exécution des programmes monitorés de plus de 50%[17].

Coûts

Une implémentation[8] utilisant ce type de mécanisme à été réalisée par K. Jain et R. Sekar et les résultats et figures que nous présentons dans cette partie sont extraits de ce travail. L'exécution d'un appel système d'un processus monitoré est représentée à la figure 2.5.

Le coût de cette implémentation a été analysé avec un corps de code d'extension vide ou qui accédait uniquement aux arguments des appels système. En effet, le coût de cette technique est principalement déterminé par la nécessité d'effectuer deux context-switch supplémentaires par appel système. Les résultats que nous présentons ici ont été obtenus avec GNU/Linux s'exécutant sur un Pentium II (350 MHz) disposant de 128MB de mémoire et d'un disque EIDE de 8GB.

La tableau 2.1 représente le coût de l'interception des appels système pour différentes applications. "CPU time" représente la somme du temps passé en user-mode et en noyau-mode au profit du processus. "Real time" représente la "dégradation" perçue réellement par l'utilisateur du système. "Low load" représente l'exécution d'une seule instance du programme (ou débit de moins de 20 Mb/s pour ftpd et httpd) alors que "High load" correspond à l'exécution de 10 instances du programme (ou débit de plus de 100 Mb/s pour ftpd et httpd).

La figure 2.6 montre l'augmentation du "CPU time" en fonction du nombre d'appels système exécutés. Ce graphe montre que le coût est compris entre 26 et 38 microsecondes par appel système avec une majorité des points concentrés autour des 34 microsecondes.

D'une manière générale, le coût pour les processus faisant un usage intensif du CPU peut être considéré comme négligeable[7], celui pour les processus faisant un usage intensif des "Disk I/O" est important d'un point de vue "CPU time" mais ne provoque pas de perceptions

Application	CPU time	Real Time (Low load)	Real time (High load)
gzip	<2%	<2%	<2%
ghostscript	10%	<5%	<10%
tar	30%	5%	10%
cp -r	50%	5%	10%
ftpd	70%	<2%	30%
httpd	65%	<5%	35%

TAB. 2.1 – Coûts de l'interception des appels système sur différentes applications

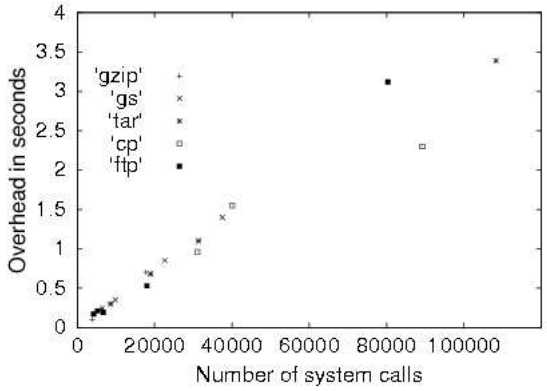

FIG. 2.6 – Augmentation du coût avec l'augmentation du nombre d'appels système

importantes de coût car la durée des opérations I/O est nettement plus importante que le "CPU time".

2.2.3 Kernel Patching

Le principe fondamental de cette méthode consiste à modifier le code source du noyau, de le recompiler et de l'installer sur les machines devant être monitorées. Le terme patching vient du fait qu'il est possible de créer un fichier (le fichier patch) contenant l'information sur tous les changements effectués par rapport au noyau d'origine et d'appliquer ce patch sur les sources d'un autre noyau non modifié afin d'obtenir les mêmes modifications.

Cette technique d'interception propose donc de modifier le code source de chaque appel système afin de lui ajouter les fonctionnalités que nous désirons (ex logging de certains paramètres), de recompiler le noyau et de l'installer sur les machines devant être monitorées.

Avantages

- Ne peut être contourné.
- Coût faible de l'interception (déterminé par le code d'extension).
- Possibilités maximum du code d'extension.

Inconvénients

- Les mécanismes de protection normale empêchant un processus d'endommager un autre processus ne sont pas présents dans le noyau-mode. Il est donc possible en cas d'erreur de programmation d'endommager un autre processus, de "planter" le système entier.
- Obligation de recompiler le nouveau noyau patché et de l'installer sur les systèmes devant être monitorés.
- Le fait de modifier le noyau pourrait être perçu comme trop risqué et ne pas être accepté.
- Peu ou pas flexible : si nous désirons modifier le comportement de l'interception, nous devons à nouveau compiler le noyau.

Coûts

Nous n'avons trouvé aucune étude contenant des informations concernant le coût de cette méthode mais nous pouvons légitimement supposer que, pour une même fonctionnalité, celui-ci ne devrait pas être supérieurs à celui provoqué par les modules.

2.2.4 Loadable Kernel Module

Les *loadable kernel modules* sont des fichiers contenant des composants du noyau pouvant être insérés et retirés dynamiquement. Bien que compilés séparément du noyau, ces modules ont accès à l'ensemble des informations qui sont exportées par le noyau et les autres modules. Ils sont la solution GNU/Linux pour atteindre d'une façon efficace la majorité des avantages théoriques du pattern microkernel[5](flexibilité, extensibilité, ...) sans introduire de dégradations au niveau des performances[15]. Ce mécanisme d'interception propose donc de développer un module qui sera ajouté au noyau des machines hôtes et qui implémentera les fonctionnalités nécessaires à l'interception des appels système. Schématiquement, les actions de ce module se présenteront de la façon suivante :

1. Introduction du module
   (a) Localisation de la **sys_call_table**
   (b) Préparation des éléments nécessaires pour effectuer les services (configuration, ...)
2. Vie du module
   (a) Pour chaque appel système devant être intercepté, effectuer le remplacement de l'appel système réel dans la **sys_call_table** par une fonction ayant la même signature mais effectuant les actions désirées (et appelant éventuellement l'appel système réel)
3. Retrait du module
   (a) Libération de toutes les ressources utilisées par le module
   (b) S'assurer que le système est laissé dans un état consistant : la **sys_call_table** doit être dans le même état qu'avant l'introduction du module

Avantages

- Ne peut être contourné.
- Coût faible de l'interception (déterminé par le code d'extension).
- Possibilités maximum du code d'extension.

Inconvénients

- Les mécanismes de protection normale empêchant un processus d'endommager un autre processus ne sont pas présents dans le mode noyau. Il est donc possible en cas d'erreur de programmation d'endommager un autre processus, voire de "planter" le système entier.
- Une attaque pourrait être exécutée avant que le code du module ne soit chargé dans le noyau.
- Un appel de fonction supplémentaire par rapport au kernel patching.
- Utilisation avec le noyau 2.6 (voir 2.3).

Coûts

Une implémentation d'un mécanisme de détection d'intrusion s'exécutant sur un *cluster* utilisant ce type de technique a été réalisé par Zhen Liu[11]. La figure 2.7[7] représente la façon dont se déroule un appel système dans son implémentation.

Les tests sont effectués sur un noyau 2.4.2. Nous devons toutefois préciser que ces tests sont réalisés sur une architecture de type cluster et représentent donc le coût pour l'ensemble du cluster. Toutefois, l'information que nous devons retenir est la différence de temps d'exécution qu'occasionne l'exécution du module. Le benchmark a donc été exécuté 20 fois suivant 3 conditions d'exécution différentes :

1. Aucune modification n'est apportée au système.
2. Le module est chargé mais ne monitore pas le programme de benchmark.
3. Le module est chargé et monitore le programme de benchmark.

[7] Les figures et tableaux utilisés dans la suite de ce point sont extraits de ce travail.

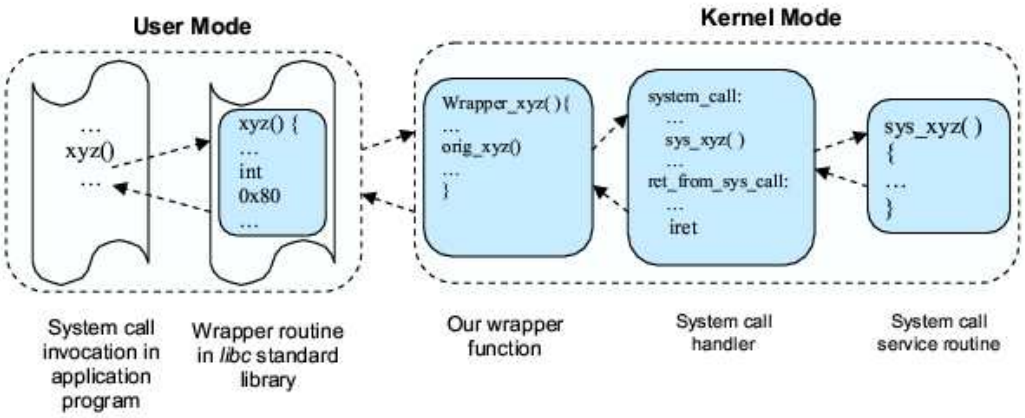

FIG. 2.7 – Exécution d'un appel système intercepté par un module

	Average Time of 20 times (seconds)	Standard Deviation
without pShield loaded	5.5475	0.06
with pShield loaded and without monitoring	5.577	0.037
with pShield loaded and monitoring	5.634	0.035

TAB. 2.2 – Résultats des tests

Le tableau 2.2 montre les résultats des différents tests. Nous remarquons que le chargement du module entraîne une augmentation du temps d'exécution de 0.53% et que le monitoring du programme de benchmark cause une augmentation de 1.55%[8].

Quelques informations plus détaillées sur les modules

Etant donné que l'implémentation que nous proposons dans ce travail est basée sur l'utilisation d'un module, il est nécessaire que nous développions plus précisément certains concepts de base relatifs aux modules. L'ensemble des points que nous présentons est principalement issu du livre "GNU/Linux Device Drivers"[16].

Un module se différencie d'une application sur les points suivants :

- Alors qu'une application exécute une seule tâche (du début à la fin), un module s'enregistre auprès du noyau afin d'offrir de nouvelles fonctionnalités à ce dernier.
- La fonction *init_ module* est l'équivalent de la fonction *main* d'une application à la différence que cette fonction se termine immédiatement. Le code exécuté pendant cette fonction permet d'initialiser les éléments nécessaires au module.

[8]Ce qui est en dessous des 5% habituellement considéré comme acceptable

- La fonction *cleanup_ module* est la fonction qui est appelée juste avant que le module ne soit enlevé du noyau et doit permettre de libérer toutes les ressources qui sont utilisées exclusivement par le noyau.
- Le développement d'un module se fait sans l'utilisation des librairies classiques. Les fonctions prédéfinies que peut utiliser un module sont uniquement celles qui sont exportées par le noyau.
- Le module s'exécute dans le mode noyau alors qu'une application s'exécute en mode utilisateur. Le principal point qu'il faut se rappeler ici est le fait que le mode noyau n'offre pas les sécurités du mode utilisateur (ex écrire dans une zone mémoire qui n'a pas été allouée au module) et que par conséquent, la programmation d'un module doit être très rigoureuse.

Un autre point particulier à la création de module est le fait qu'il faut être très attentif aux impacts de la concurrence. Le code d'un module doit également permettre la "réentrance" et l'accès à des ressources communes doit donc généralement se faire en utilisant des locks et des sémaphores. En effet, c'est l'utilisation correcte de ces mécanismes qui permettra à une application de ne pas avoir d'effets de *race condition*. Il est également utile de rappeler qu'un code utilisant des mécanismes de locks et de sémaphores ne doit pas pouvoir engendrer des situations de *deadlocks*.

Lorsqu'un module exécute des instructions, il le fait au profit d'un autre processus s'exécutant en mode utilisateur. Le module peut accéder aux informations relatives au processus ayant provoqué son exécution en accédant à la structure **current** qui contient tous les paramètres du processus s'exécutant actuellement.

Un module ne peut être retiré du noyau que s'il laissera celui-ci dans un état consistant (ie : retrait uniquement quand le module n'est pas utilisé). Afin de permettre au programmeur de s'assurer que le module ne sera pas enlevé à un moment inopportun, il est possible de tenir compte manuellement de l'utilisation du module (c'est lorsque cette utilisation est égale à 0 que le retrait est possible) en travaillant avec les trois macros suivantes :

- MOD_INC_USE_COUNT : incrémente l'usage du module
- MOD_DEC_USE_COUNT : décrémente l'usage du module
- MOD_IN_USE : évaluer à true si l'usage est différent de 0.

2.3 Vers le noyau 2.6

Le noyau GNU/Linux étant en constante évolution, il est intéressant de se rendre compte de la mesure dans laquelle les techniques que nous avons exposées pour le noyau 2.4 seront encore d'application lors de la migration vers un noyau de type 2.6.

Si les trois premières techniques que nous avons présentées seront toujours réalisables avec le noyau 2.6, l'interception des appels système en utilisant un module du noyau ne sera plus réalisable directement car, dans un souci de sécurité, les concepteurs du noyau 2.6 n'ont plus permis à des modules de réaliser le *hijacking* des appels système et, par conséquent, ces derniers n'auront plus accès directement à la **sys_call_table**[9]. Toutefois, il est toujours possible d'accéder à cette table d'une façon indirecte :

- Une première possibilité consiste à modifier les sources du noyau 2.6 et d'ajouter l'exportation de la **sys_call_table** et donc de la rendre à nouveau accessible à l'ensemble des composants du noyau.

[9]Nous ne pouvons toutefois pas affirmer que ce sera le cas pour la totalité des noyaux 2.6.

– Une seconde solution consiste à réutiliser le code de différents modules ayant déjà contourné le problème (ARLA[10], Snare[11]).

Nous pouvons donc conclure que l'ensemble des techniques que nous avons développées dans ce chapitre pourront trouver une application lors de l'évolution vers un noyau 2.6.

[10] `http://www.stacken.kth.se/projekt/arla/`

[11] `http://www.intersectalliance.com/projects/Snare/`

Chapitre 3

Description de divers systèmes existants pertinents pour les objectifs du mémoire

3.1 Les standards C2 et Common criteria

Il est important d'avoir des systèmes informatiques sécurisés et également de définir précisément la notion de système d'exploitation sécurisé. C'est pour cette raison que des documents ont été dévéloppés afin d'établir les caractéristiques nécessaires pour affirmer qu'un système d'exploitation est sécurisé.

Le standard C2

Le département de la Défense Nord Américain a publié un document sur la sécurité des systèmes d'exploitation qui est souvent appelé *The orange book* [14]. Ce livre date de décembre 1985 et classe les systèmes d'exploitation en sept catégories qui sont fonction de leurs caractéristiques de sécurité (D, C1, C2, B1, B2, B3, A1). Ce standard est à présent remplacé par une nouvelle version appelée Common Criteria, plus complexe mais également plus internationale. Ce livre reste toutefois extrêmement intéressant. Nous parlerons plus particulièrement de la classe appelée C2 qui est la classe de base pour l'acceptation d'utilisation d'un système d'exploitation dans le domaine critique tel que l'armée ou les administrations publiques.

La classe C2 exige un certain nombre de caractéristiques en matière de sécurité. La liste [19] suivante présente une vue d'ensemble des caractéristiques prévues par *The orange book* :

1. Politique de sécurité
   (a) Contrôle d'accès discrétionnaire
   (b) Réutilisation des objets
2. Fiabilité
   (a) Identification et authentification
   (b) Audit
3. Surveillance

(a) Architecture système

(b) Intégrité du système

4. Documentation

(a) Guide des caractéristiques de la sécurité

(b) Manuel des installations approuvées

(c) Documentation des tests

(d) Documentation de la conception.

La caractéristique de sécurité "Audit" nous intéresse plus particulièrement car elle est le sujet de notre mémoire. Nous nous limiterons uniquement au développement de cette caratéristique. *The orange book* définit un ensemble de fonctionnalités de base concernant les caractéristiques d'audit :

- L'audit devrait être capable de créer, de maintenir et de protéger les *audit trails* contre les modifications, accès non autorisés et destructions d'une trace d'audit.
- Les données auditées devrait être protégées par le système d'exploitation limitant le nombre d'utilisateurs ayant l'autorisation de les lire.
- Le système d'exploitation devrait être capable d'enregistrer les événements suivants :
  - Identification et authentification
  - Ouverture de fichier
  - Exécution de programme
  - Introduction/effacement d'un objet dans l'espace utilisateur
  - Actions réalisées par un administrateur
  - Tout autre événement relatif à la sécurité
- Les enregistrements d'événement doivent être composés :
  - De la date et de l'heure de l'événement
  - De l'utilisateur qui a réalisé l'événement
  - Le type d'événement
  - L'information si l'exécution de l'événement s'est réalisé avec succès
  - Pour l'identification/authentification : l'origine de la requête
  - Pour l'introduction d'un objet ou l'effacement d'un objet : le nom de l'objet
- Le système d'exploitation doit être capable d'auditer sélectivement les actions d'un ou de plusieurs utilisateurs.

Common Criteria

Common criteria [6] (CC) est un nouveau standard international ISO/IEC 15408 :1999. CC constitue la mise en commun de différents travaux à travers le monde sur les caractéristiques des technologies de la sécurité et est reconnu par une grande majorité des institutions de sécurité des pays (France, Etat Unis, ...). CC utilise un langage plus précis et plus formel que C2.

Dans l'analyse qui suit, nous nous limiterons également aux caractéristiques concernant l'audit des systèmes. Le terme audit englobe ici la reconnaissance, l'enregistrement et l'analyse d'information d'activité relevant de la sécurité. CC prévoit un ensemble de fonctionnalités [20] concernant les caractéristiques d'audit :

- Définition d'une réponse automatique dans le cas d'une détection d'un événement potentiellement en violation de la sécurité

- Définition des événements à auditer par le système :
  - Démarrage et arrêt du système
  - Tous les événements minimum pour le niveau audit de base
  - Tous les événements spécifiques
- Les enregistrements doivent au moins contenir les informations suivantes :
  - Date et heure, type de l'événement, utilisateur qui a réalisé l'événement, réussite/erreur de l'événement
- Définition des exigences pour une analyse de l'activité du système pour la détection de violation possible du système.
- Le système doit être capable d'appliquer un ensemble de règles analysant l'audit des événements. Les règles doivent être une accumulation ou une combinaison de connaissances indiquant une potentielle violation.
- Limitation des utilisateurs ayant la capacité de lire ou d'interpréter l'audit.
- Le système doit être capable d'inclure ou d'exclure des événements de l'audit en fonction de l'identité de l'objet, d'un utilisateur, d'un sujet, d'un hôte ou d'un type d'événement.
- Le système doit être capable de protéger de l'effacement non autorisé du stockage des enregistrements d'audit.
- Le système doit être capable de prévenir ou de détecter une modification des enregistrements.
- Le système doit prévoir une action au cas où la trace d'audit excède la taille prédéfinie.

Remarque

Il peut être constaté que les exigences de Common Criteria sont mieux définies et plus précises que celles du standard C2. Nous avons maintenant une vue globale des caractéristiques nécessaires pour la réalisation d'un système d'audit. De plus l'analyse de Common Criteria et de C2 servira de guide dans la réalisation de notre système d'audit.

3.2 BSM

3.2.1 Présentation succincte

BSM [12](Basic Security module) est un module réalisé par la société Sun Microsystems pour son système d'exploitation UNIX Solaris. Cette application offre la possibilité de logger l'activité du système. Ce système d'audit fournit à Solaris les caractéristiques nécessaires pour obtenir le niveau de sécurité C2.

Le but de l'étude de BSM est principalement d'analyser un système d'audit existant et utilisé, afin d'en extraire les points intéressants que nous pourrions incorporer dans notre réalisation.

3.2.2 Architecture

Nous n'avons pas trouvé de document nous fournissant d'informations sur l'implémentation de BSM et, étant donné que notre réalisation est destinée au système GNU/Linux, nous n'avons pas effectué plus de recherches à ce sujet. Notre analyse portera donc principalement sur les fonctionnalités fournies aux utilisateurs ainsi que sur le système de configuration.

BSM est lancé sur une machine à partir de la commande *bsmconv*. La configuration de BSM s'effectue en utilisant quatre fichiers se trouvant dans le répertoire /etc/security :

- Le fichier audit_class définit les classes d'événements : chaque classe est composée d'un ou de plusieurs événements.
- Le fichier audit_event définit les événements auditables : l'ensemble des appels système est auditable mais également une série d'applications utilisateur.
- Le fichier audit_control définit le paramétrage du système pour l'audit des classes : différents paramètres sont possibles (ex : le fichier de stockage des enregistrements, le minimum d'espace libre, les différentes classes d'événements à auditer). Pour cette partie, l'ensemble des utilisateurs seront audités.
- Le fichier audit_user définit les classes à auditer spécifiquement pour chaque utilisateur.

On peut donc avec BSM configurer de manière très précise l'ensemble des événements à auditer. La configuration se fait uniquement en sélectionnant les classes à auditer dans le fichier audit_control ou dans le fichier audit_user. L'utilisation de la notion de classe dans la configuration permet de rassembler logiquement des événements entre eux, par exemple il est possible de rassembler tous les événements relatifs à la gestion des fichiers (open, read, close ...) dans une classe. Cela simplifie la configuration de l'outil par l'administrateur qui, en règle générale, cherche à auditer un groupe d'événements.

Chaque enregistrement décrit un événement et contient les informations suivantes :

- L'initiateur de l'action
- Les fichiers qui ont été affectés
- Les actions qui ont été essayées
- Le moment et la localisation de l'action.

L'*audit trail* est créé par le démon **auditd** qui se charge de collecter les enregistrements et de les sauver dans un fichier. Deux outils sont mis à la disposition des utilisateurs de BSM pour les aider à analyser les enregistrements, **praudit** et **auditreduce**. La commande **praudit** est une commande spécifique pour l'administrateur, elle lit un *audit trail* et affiche l'information des enregistrements dans un format ASCII compréhensible pour un administrateur. La commande **auditreduce** permet de sélectionner ou de fusionner des enregistrements d'audit (les *audit trails* pouvant provenir d'une ou de plusieurs machines). La sélection peut se faire à partir de différents critères tels que la date, l'heure, l'utilisateur, la classe d'audit.

Il faut aussi noter que BSM ne permet pas de personnaliser l'information de l'audit car les informations sont prédéfinies pour chaque événement. Notons également qu'ASAX peut être utilisé pour analyser les *audit trails* BSM.

3.2.3 Utilité dans le cadre de ce travail

Le système BSM utilise un système de configuration multi-fichiers assez intéressant pouvant servir d'inspiration dans la réalisation de notre système de configuration. Ce type de configuration permet une approche plus structurée en associant chacun des fichiers à un concept différent.

La notion de classe d'événements est très importante étant donné qu'elle fournit une facilité de configuration aux utilisateurs du système d'audit : le principe est de regrouper dans une classe l'ensemble des appels système ayant une certaine relation (e.g. classe reprenant toutes les actions relatives aux accès de fichiers (open, close, write, read), classe reprenant toutes les opérations sur la création et destruction de processus (fork, exec, exit)).

L'utilisation d'un démon qui se charge de récolter les enregistrements et de les logger dans des fichiers est également une idée intéressante. Le démon est un logiciel constamment

en activité qui est exécuté directement au démarrage de l'ordinateur. Le système d'audit est donc activé de manière automatique dès l' initialisation du système d'exploitation.

3.3 Syscalltrack

3.3.1 Présentation succincte

Syscalltrack[1] est un produit permettant à un "superuser" d'intercepter les appels système sur GNU/Linux dans le but premier d'effectuer du debugging. La technique utilisée pour intercepter les appels système est celle des "Loadable Kernel Module". Ce logiciel est principalement développé sous la license GNU[2] et une partie (e.g. les librairies) est développée sous la license GNU LGPL[3]. Ce logiciel permet donc d'effectuer l'interception de certains appels système et l'exécution d'actions supplémentaires lors de cette interception si certaines conditions (définies par l'utilisateur) sont validées. Toute la paramétrisation du logiciel s'effectue via un fichier de configuration ayant une syntaxe bien définie. Une action typique prévue est l'envoi des informations interceptées sur un device file qui peut ensuite être lu par différents programmes.

3.3.2 Architecture

Le logiciel Syscalltrack se décompose en trois parties distinctes :

1. Un module "hijacker" responsable de l'interception des appels système.
2. Un module "filtre" responsable de déterminer si l'utilisateur est intéressé par l'appel système en cours et le cas échéant, de transmettre les informations demandées en mode utilisateur.
3. Une librairie de communication permettant à un processus se trouvant en mode utilisateur d'effectuer la configuration du module.

Le module hijacker

Ce module est responsable d'intercepter les appels système et d'initier les actions désirées durant cette interception. L'implémentation du hijacking est réalisée suivant la procédure générale que nous avons introduite dans le chapitre précédent : lorsqu'un appel système doit être hijacké, la position de cet appel système est déterminée dans la **syscall_table.** L'appel système est alors remplacé par une nouvelle fonction ayant la même signature (fonction stub) et réalisant les actions supplémentaires. Ce module permet d'effectuer des actions avant l'exécution de l'appel système réel ainsi que des actions après l'exécution de l'appel système réel. Il peut également être considéré comme relativement indépendant du reste du projet car il n'utilise aucune des autres parties de celui-ci et exporte, au niveau du noyau dans lequel il est déployé, les fonctionnalités suivantes :

- hijack_syscall_before : l'utilisation de cette fonction permet de définir les opérations qui doivent être exécutées avant l'invocation de l'appel système original
- hijack_syscall_after : l'utilisation de cette fonction permet de définir les opérations qui doivent être exécutées après l'exécution de l'appel système original

[1] http ://Syscalltrack.sourceforge.net/

[2] General Public License : http ://cvs.sourceforge.net/viewcvs.py/*checkout*/Syscalltrack/Syscalltrack/COPYING

[3] Lesser General Public License

- release_syscall_before : l'utilisation de cette fonction permet d'arrêter l'exécution des opérations qui avaient été définies lors d'un précédent appel de la fonction hijack_syscall_before
- release_syscall_after : l'utilisation de cette fonction permet d'arrêter l'exécution des opérations qui avaient été définies lors d'un précédent appel de la fonction hijack_syscall_after.

Le module "filtre"

Le logiciel a été réalisé dans le but d'effectuer du debugging. Etant donné que les appels système sont hijackés au niveau du système (et que l'utilisateur n'est pas forcément intéressé pour avoir une information sur la totalité des invocations des appels système qu'il a hijackés mais peut-être uniquement sous certaines conditions (accès vers un fichier bien spécifique)), il fallait avoir un composant décidant si les actions à réaliser avant ou après l'invocation réelle de l'appel système doivent être exécutées : c'est une des responsabilités de ce module. Lorsque le fichier de configuration du logiciel est correct, c'est ce module qui appellera les fonctions exportées par le module "hijacker" afin d'intercepter les appels système désirés. Une responsabilité supplémentaire du module est la configuration du logiciel suivant les directives fournies par l'utilisateur.

La librairie de communication

Une librairie de communication spécifique à ce logiciel et utilisant les fonctions *sysctl()* et *ioctl()* a été développée afin de permettre la communication entre des processus s'exécutant en mode utilisateur et les modules de Syscalltrack. La fonction *sysctl()* permet de contrôler des paramètres du noyau durant son exécution et la fonction *ioctl()* permet à un processus s'exécutant en mode utilisateur de configurer un module.

3.3.3 Utilité dans le cadre de ce travail

L'ensemble du projet est utile dans la réalisation de ce mémoire car les auteurs ont effectué un effort considérable au niveau de la documentation. De même, la mailing list de Syscalltrack est active et les principaux acteurs du projet apportent souvent leurs idées lorqu'une question est posée (même si elle n'a qu'un rapport éloigné avec leur projet). Si nous nous attachons à l'aspect réutilisation de leur travail, nous pouvons à nouveau décomposer l'ensemble du projet en trois grandes parties.

Le module hijacker

Le code constituant ce module est la partie la plus intéressante. Comme nous l'avons mentionné plus haut, cette partie est relativement indépendante de l'ensemble du projet et peut donc servir de base à un nouveau logiciel. Ce point est particulièrement intéressant car un nombre important de problèmes typiques lors de la création d'un module d'hijacking d'appels système sont résolus :

- Impossibilité de retirer le module alors qu'un appel système est encore hijacké : ceci implique que le noyau sera laissé dans un état consistant (en tout cas, en ce qui concerne les structures relatives aux appels système).
- Impossibilité de retirer une fonction stub alors qu'un appel de cette fonction est encore en cours (nous reviendrons plus précisément sur ce point dans le chapitre relatif à l'implémentation de notre module)

Les actions exécutées lorsqu'un appel système est hijacké sont identiques pour chaque appel système :

1. Exécution des instructions avant l'appel système original
2. Exécution de l'appel original
3. Exécution des instructions après l'appel système original

Les concepteurs de ce logiciel ont tiré parti de cette particularité et ont développé un script perl permettant une génération automatique du code relatif aux fonctions stubs à partir d'un fichier contenant la signature des différents appels système. Ce script peut servir de base à la création d'un nouveau script répondant précisément à nos besoins.

Les fonctions exportées par le module ne sont pas directement intéressantes dans le cadre de ce travail car nous ne recherchons pas la possibilité d'effectuer le hijacking avant ou après l'exécution de l'appel système original. Toutefois, les mécanismes généraux de remplacement d'un appel système qui sont implémentés dans ces fonctions sont semblables à ceux que nous devrons développer.

Le module filtre

Si ce module était tout à fait judicieux dans le cadre de Syscalltrack, il n'est plus nécessaire dans le cadre de notre travail. En effet, ce filtre engendre un coût au niveau des performances du système et est redondant avec la fonction assurée par ASAX (qui effectue également un filtre au niveau des informations d'audit). Nous pourrions toutefois l'adapter et l'exploiter comme mécanisme de préfiltrage afin qu'ASAX ait moins d'informations à traiter. Cette option a retenu notre attention durant une certaine période mais cette réutilisation a pour effet de réduire considérablement la simplicité du code et d'aller à l'encontre de certains objectifs que nous poursuivions (simplicité du code, ajout aisé de nouvelles fonctionnalités). Nous avons donc décidé de ne pas réutiliser cette partie du projet.

La librairie de communication

Cette partie est spécifique à Syscalltrack et nous a été utile afin d'être en possession d'un exemple concret d'implémentation de certains mécanismes de communication.

3.3.4 Eléments manquants dans le cadre de ce travail

Si ce module peut servir de base pour la réalisation d'une partie importante des objectifs de notre travail, il reste des aspects pour lesquels il n'apporte pas de solution. Le premier de ceux-ci est le fait que l'objectif de Syscalltrack est d'effectuer du debugging et qu'en conséquence, il n'apporte pas de solutions sécurisées (déchargement du module, tout le monde peut lire sur le *device file* de sortie, toutes les informations interceptées avant qu'un processus ne commence à lire sur le *device file* sont perdues, ...). Le deuxième problème est que les informations lues sur la sortie sont des strings, ce qui signifie que nous devons effectuer une transformation de la sortie afin d'obtenir une sortie de type NADF.

3.4 SNARE

3.4.1 Présentation succincte

SNARE[4] est un projet développé par IntersectAlliance pour plusieurs operating systems différents. Contrairement à Syscalltrack, le but de SNARE est de rendre possible une détection d'intrusion sur divers système d'exploitation. La partie que nous développons à présent est distribuée sous la "GNU Public License" mais les outils nécessaires pour exploiter les informations auditées (ie : le SNARE server) ne sont pas distribués librement.

3.4.2 Architecture

Le projet se divise en trois parties distinctes :

1. Un module responsable de l'interception des appels système (ce module peut être remplacé par un patch du noyau)
2. Un démon s'exécutant en mode utilisateur et responsable de configurer le module, d'extraire l'information disponible dans le module et d'effectuer le filtrage demandé par l'utilisateur
3. Une interface graphique de communication qui montre les événements sélectionnés par l'utilisateur lorsqu'ils surviennent.

Le module

Le code du module est encore une fois, dans le cadre de notre travail, la partie la plus intéressante du projet. Le mécanisme de hijacking utilisé est le mécanisme commun pour ce genre de manipulations (cf 4.1.1). Les principales différences relatives au hijacking que nous avons trouvées par rapport à Syscalltrack sont les suivantes :

- SNARE ne permet d'hijacker que les appels système jugés intéressants par ses concepteurs.
- Chaque appel système pouvant être hijacké doit être codé séparément. Toutefois, les actions qui sont communes à l'ensemble des appels système sont codées dans des fonctions spécifiques qui sont appelées dans le code spécifique à chaque appel système.
- Il n'est pas possible de choisir le moment où seront effectuées les actions de hijacking (i.e. avant ou après l'exécution de l'appel système original), ni de modifier le format des informations loggées.
- Il est possible de laisser le noyau dans un état inconsistant lorsque le module est retiré : cela signifie qu'il est possible de retirer le module du noyau alors qu'une partie du code du module est susceptible d'être encore exécuté. Une explication détaillée de ce problème est présent dans la partie 4.1.3 et nous nous contentons à présent d'exposer les principes du problème : la situation est susceptible de se produire lorsqu'une fonction stub invoque un appel système original (A) mais que celui-ci ne retourne pas directement. Il est alors possible que le module soit retiré du noyau avant le retour de l'appel système et l'exécution de la fin du code de la fonction stub. Au moment où l'appel système (A) se terminera, le noyau tentera d'exécuter le code situé dans l'espace mémoire qui était alloué au module et qui contenait la suite des instructions de la fonction stub. Or, comme le module n'est plus présent, le noyau tentera d'exécuter les instructions

[4]http ://www.intersectalliance.com/projects/SNARE

se trouvant dans cette même zone mémoire et qui pourraient être encore présentes si, par chance, cet espace mémoire n'a pas été réalloué par le noyau ou qui pourraient être complètement aléatoires si cet espace mémoire a été alloué à un nouveau processus.

Le module communique les informations d'audit via un fichier '/proc/audit' et prévient le démon de la disponibilité de ces informations en lui envoyant un signal SIGIO (le module considère que le premier processus ouvrant le fichier '/proc/audit' est le démon).

Le démon

Le démon permet d'effectuer la configuration du module afin de sélectionner les appels système devant être hijackés en accord avec le contenu des fichiers de configuration. Lorsque le démon réceptionne des données d'audit, celles-ci sont dans un format binaire et il les convertit alors vers un format texte. Cette information est généralement stockée dans un fichier.

Interface graphique

Afin de permettre une configuration aisée du logiciel, une interface graphique est également disponible. Elle permet aussi de consulter les informations présentes dans le fichier stockant les informations déjà interceptées.

3.4.3 Utilité dans le cadre de ce travail

Ce projet constitue une excellente base afin de développer des mécanismes de communication entre un processus s'exécutant en mode utilisateur et un module s'exécutant en mode noyau. Si Syscalltrack donnait déjà des exemples de communication dans le sens mode utilisateur vers mode noyau, SNARE donne des exemples de communications efficaces dans le sens inverse. Du fait de sa vocation de sécurité, SNARE constitue également un exemple pour l'implémentation d'un mécanisme d'audit utilisant des buffers afin de ne pas perdre des événements (ou d'être au courant que des événements sont perdus). Enfin, l'architecture globale du projet peut être réexploitée afin de développer un nouveau système d'audit.

3.4.4 Eléments manquants dans le cadre de ce travail

L'élément le plus important qui manque dans ce projet est l'absence de sécurité suffisante pour que le kernel soit toujours laissé dans un état stable lorsque le module est retiré. Il est en effet possible (en réalisant certaines manipulations) de retirer le module du kernel alors que certaines parties du code de module doivent encore être exécutées. Le but de ce chapitre étant uniquement de présenter certains outils existants, nous renvoyons à la partie 4.1.3 pour avoir une explication plus approfondie.

Le second élément manquant est le fait qu'il n'est pas possible d'hijacker l'ensemble des appels système et que l'information extraite d'un événement n'est pas configurable.

Un troisième et dernier élément manquant est le fait que le format de sortie n'est pas du NADF.

Chapitre 4

Implémentation d'un mécanisme d'audit pour GNU/Linux, adapté à ASAX

4.1 Implémentation du module

L'implémentation des mécanismes internes de notre module est fortement influencée par le logiciel Syscalltrack. Dans un souci d'efficacité, nous avons basé notre implémentation sur celle de leur module "hijacker". Dans un premier temps, nous l'avons donc analysé et "nettoyé"[1]. Lorsque cette étape a été réalisée, nous possédions un squelette fonctionnel du module. Nous avons alors ajouté les parties qui étaient manquantes :

- Ajout de nouveaux champs dans les structures déjà présentes (eg le champs format de la structure syscall_hijack_data qui maintient une représentation des informations qui doivent être présentes dans le *record* NADF)
- Algorithme propre à notre application
- Buffering des informations interceptées (influence de SNARE)
- Communication avec le démon afin de rendre l'information disponible en user mode (influence de SNARE)
- Pas de logging pour les actions du démon (sauf si retrait suivi de la remise en action du démon).

4.1.1 Mécanisme de hijacking

Idée générale du mécanisme

Afin de hijacker un appel système, nous devons remplacer le pointeur se trouvant dans la table **syscall_table** référençant cet appel système par un pointeur référençant notre fonction stub (qui doit avoir la même signature). En toute généralité, nous désirons donc modifier l'état initial du kernel représenté à la figure 4.1 par l'état représenté à la figure 4.2.

[1] Le terme nettoyer ne doit pas être compris dans un sens péjoratif mais est utilisé pour indiquer que nous avons retiré toutes les instructions et structures que nous avons jugées trop spécifiques à Syscalltrack.

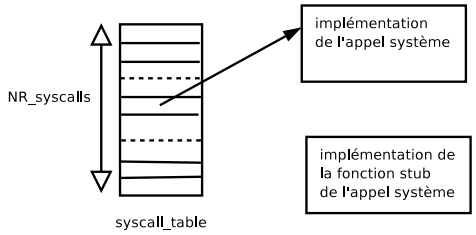

FIG. 4.1 – Etat du kernel avant le hijacking

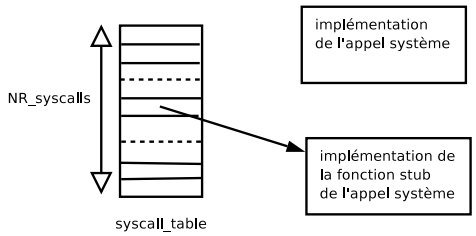

FIG. 4.2 – Etat du kernel après le hijacking

Structures utilisées

Afin de pouvoir maintenir les informations relatives à chaque appel système, nous avons utilisé la structure suivante :

```
struct syscall_hijack_data {
  func_ptr orig_syscall;
  struct semaphore sem;
  atomic_t invocations;
  int defered_release_flag;
  char* format;
};
```

Les différents éléments composant cette structure sont :

- *orig_ syscall* est un pointeur vers la fonction implémentant l'appel système original. Nous avons besoin de maintenir cette information car nous voulons être capables d'invoquer l'appel système original lorsque nous exécutons notre fonction stub mais surtout pour pouvoir laisser le kernel dans le même état qu'il était avant le hijacking lorsque nous décidons d'arrêter le hijacking.
- *sem* est un sémaphore que nous utilisons lorsque nous exécutons certaines opérations sur la structure.
- *invocations* est un compteur qui maintient le nombre d'invocations actuelles de la fonction stub. Cette information est la première des deux informations primordiales permettant d'éviter de laisser le kernel dans un état inconsistant lorsque nous désirons enlever le module[2].
- *defered_ release_ flag* est un flag indiquant si une demande d'arrêt de hijacking de cet appel système a été demandé mais que celui-ci n'a pu être réalisé pour des raisons de consistance (ie. au moment de la demande, des invocations du stub de cet appel système étaient en cours). Il s'agit donc de la seconde information permettant de ne pas laisser le kernel dans un état inconsistant lorsque nous désirons enlever le module.
- *format* est une chaîne de caractères permettant de déterminer les informations qui doivent être présentes dans le *record* NADF.

Nous avons de plus utilisé un tableau de **NR_syscalls** entrées (*hijack_ data*) afin de maintenir une structure *syscall_ hijack_ data* par appel système. Logiquement, la structure se trouvant à l'index x du tableau *hijack_ data* correspond aux informations relatives à l'appel système se trouvant à la position x du tableau **syscall_table**.

Implémentation du hijacking

Les informations fournies au module lors d'une demande de hijacking sont l'index dans le **syscall_table** de l'appel système à hijacker ainsi qu'une chaîne de caractères représentant le contenu des informations désirées par l'utilisateur. Une fois que ces arguments sont connus, l'algorithme décrit à la figure 4.4 est exécuté[3]. La spécification de cette fonction est présentée à la figure 4.3.

Nous expliquons à présent les différents points de passage de celui-ci :

[2]Une explication plus complète sur ce problème sera développée dans les points suivants.

[3]La convention que nous utilisons est que si un test est positif dans un noeud de décision (losange) alors nous suivons le chemin vers le bas

```
BUT : remplacer un appel système par un stub
PRE : syscall_id est un index représentant la position dans la syscall_table
          de l'appel système à remplacer
     format est un pointeur vers une chaîne de caractère définissant les
          informations que nous désirons   trouver dans le log à propos de
          cet appel système
POST :-SI syscall_id < 0 | syscall_id >= NR_syscalls
        ALORS -ENOSYS est retourné et rien n'est modifié
     -SI un signal est réceptionné pendant l'obtention du sémaphore de la
         structure syscall_hijack_data[syscall_id]
        ALORS -ERESTARTSYS est retourné et rien n'est modifié
     -SI le syscall a été précédemment hijacké
        ALORS 1 est retourné et rien n'est modifié
     -SI aucune stub_function n'existe pour le syscall
        ALORS -ENOENT est retourné et rien n'est modifié
     -SI aucune des conditions précédentes n'est vrai
        ALORS le syscall ayant l'index syscall_id est hijacké
              ET syscall_hijack_data[syscall_id]->format = format
              ET syscall_hijack_data[syscall_id]->defered_release_flag = 0
              ET syscall_hijack_data[syscall_id]->orig_syscall pointe vers
                 l'appel système original
              ET le "usage_count" du module est adapté afin d'empêcher un
                 retrait du module tant que le hijacking est actif
              ET 0 est retourné

int hijack_syscall(int syscall_id,char* format);
```

FIG. 4.3 – Spécification de la fonction hijack_syscall

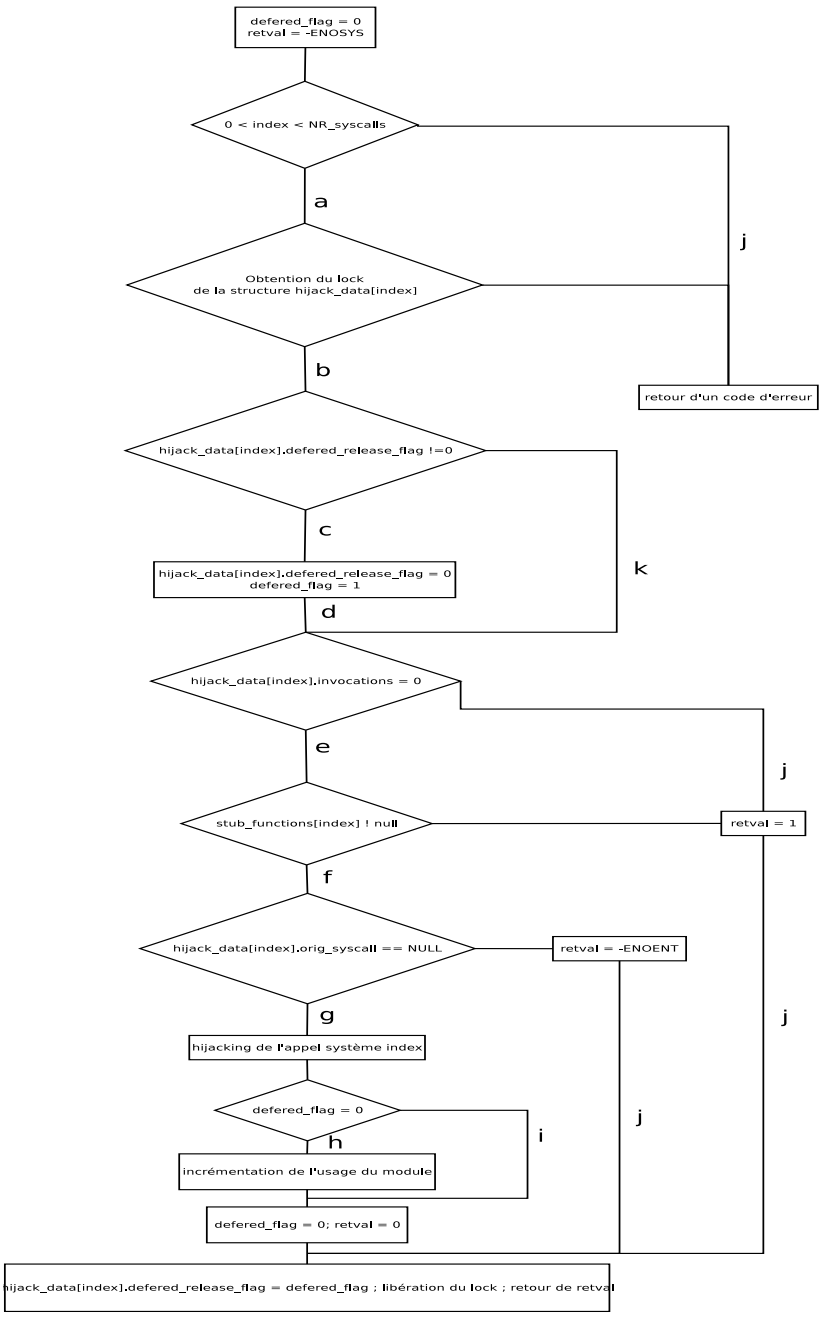

FIG. 4.4 – Algorithme de hijacking

- a : la **syscall_table** a bien été localisée et l'index de l'appel système qui doit être hijacké est compris entre 0 et **nr_syscalls**
- b : la structure syscall_hijack_data relative à l'appel système a pu être lockée
- c : le hijacking de cet appel système a déjà été réalisé précédemment et une demande d'arrêt de hijacking a été requise mais n'a pas été exécutée (voir 4.1.1)
- d : defered_flag prend la valeur 1 afin que nous puissions nous rappeler qu'un arrêt de hijacking non exécuté s'est produit pour cet appel système et que le champ defered_release_flag de la structure syscall_hijack_data est replacé à 0 (les raisons de ces manipulations deviendront évidentes lors de la lecture de 4.1.1)
- e : aucune invocation de la fonction stub n'est en cours
- f : une fonction stub existe bien pour cet appel système
- g : c'est bien l'implémentation originale[4] de l'appel système qui se trouve dans la **syscall_table**
- h : nous incrémentons l'usage du module afin d'interdire le retrait de celui-ci tant que le hijacking est effectif
- i : comme une demande de fin de hijacking n'a pas été effectuée précédemment, nous ne devons pas incrémenter l'usage du module car il n'a pas été décrémenté depuis le précédent hijacking
- j : un problème est survenu, nous ne modifions rien
- k : il n'y a pas eu de demande non exécutée de suppression de hijacking

Après que le hijacking ait été effectué, nous obtenons une situation similaire à celle de la figure 4.2. Certains éléments de notre implémentation étant absents de cette dernière, la figure 4.5 montre la situation spécifique à notre implémentation et permet de se rendre compte qu'une référence vers l'appel système original est effectivement maintenue au niveau de la structure *syscall_hijack_data* et que c'est grâce à cette structure que notre fonction stub est capable d'invoquer l'appel système original.

Une spécification de cette fonction se trouve en annexe E.

Implémentation du retrait du hijacking

L' information requise pour arrêter le hijacking d'un appel système est l'index de celui-ci dans la **syscall_table**. L'algorithme exécuté est présenté à la figure 4.7 et sa spécification à la figure 4.6.

Comme pour l'algorithme de hijacking, nous expliquons à présent les différents points de passage :

- a : l'index de l'appel système qui doit être hijacké est compris entre 0 et **nr_syscalls**
- b : la structure syscall_hijack_data relative à l'appel système a pu être lockée
- c : l'appel système était toujours hijacké
- d : le hijacking de l'appel système est terminé (ie tout appel futur de cet appel système ne sera plus intercepté)
- e : le stub n'est plus utilisé, nous pouvons donc décrémenter l'usage du kernel (si c'était le dernier appel système hijacké, il devient donc théoriquement possible de retirer le module)
- f : un problème est survenu, nous ne modifions rien
- g : l'appel système n'était plus hijacké, nous ne modifions rien

[4]En fait, nous sommes certains que ce n'est pas une de nos fonctions stub mais il se pourrait que ce soit un stub indépendant de notre module

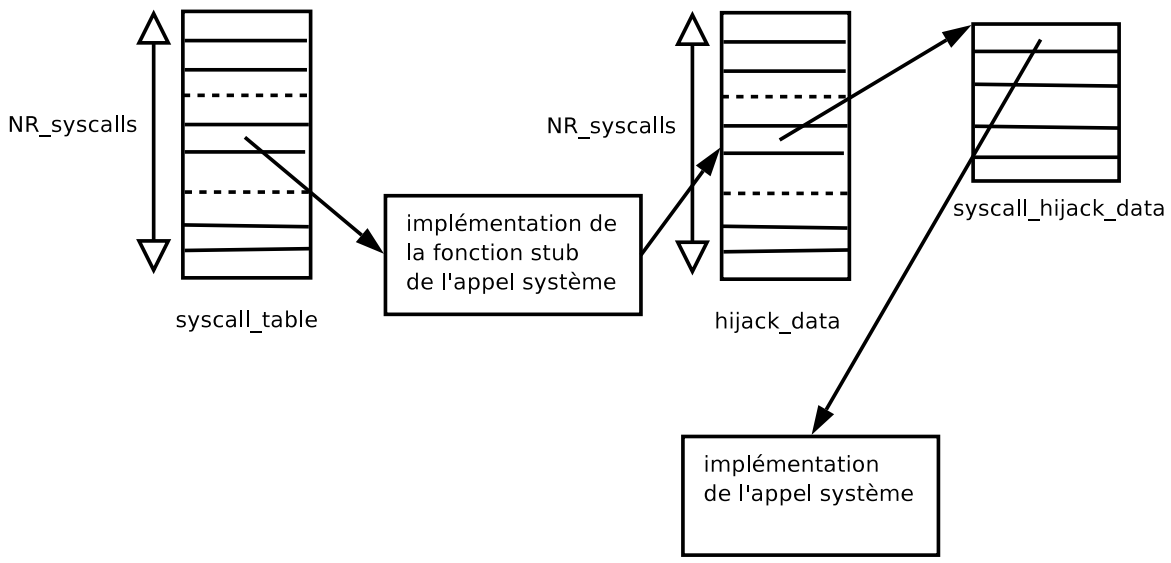

FIG. 4.5 – Références des éléments de notre implémentation après hijacking

```
BUT : Terminer le hijacking d'un appel système
PRE :  syscall_id est un index représentant la position dans la syscall_table
         de l'appel système à replacer
POST :-SI syscall_id < 0 | syscall_id >= NR_syscalls
        ALORS -ENOSYS est retourné et rien n'est modifié
     -SI un signal est réceptionné pendant l'obtention du sémaphore de la
        structure  syscall_hijack_data[syscall_id]
        ALORS -ERESTARTSYS est retourné et rien n'est modifié
     -SI aucun hijacking n'est actif sur cet appel système
        ALORS 0 est retourné et rien n'est modifié
     -SI l'appel système était hijacké et que le stub n'est pas utilisé
        ALORS syscall_hijack_data[syscall_id]->orig_syscall est replacé dans
        la syscall_table à l'index syscall_id
              ET le 'usage count" du module est diminué de 1
              ET l'espace mémoire alloué à
                 syscall_hijack_data[syscall_id]->format est libérée
     -SI l'appel système était hijacké et que syscall_hijack_data[syscall_id]->invocations
        ALORS syscall_hijack_data[syscall_id]->orig_syscall
              est replacé dans la syscall_table à l'index syscall_id
              ET syscall_hijack_data[syscall_id]->defered_release_flag = 1
int release_syscall(int syscall_id);
```

FIG. 4.6 – Spécification de la fonction release_syscall

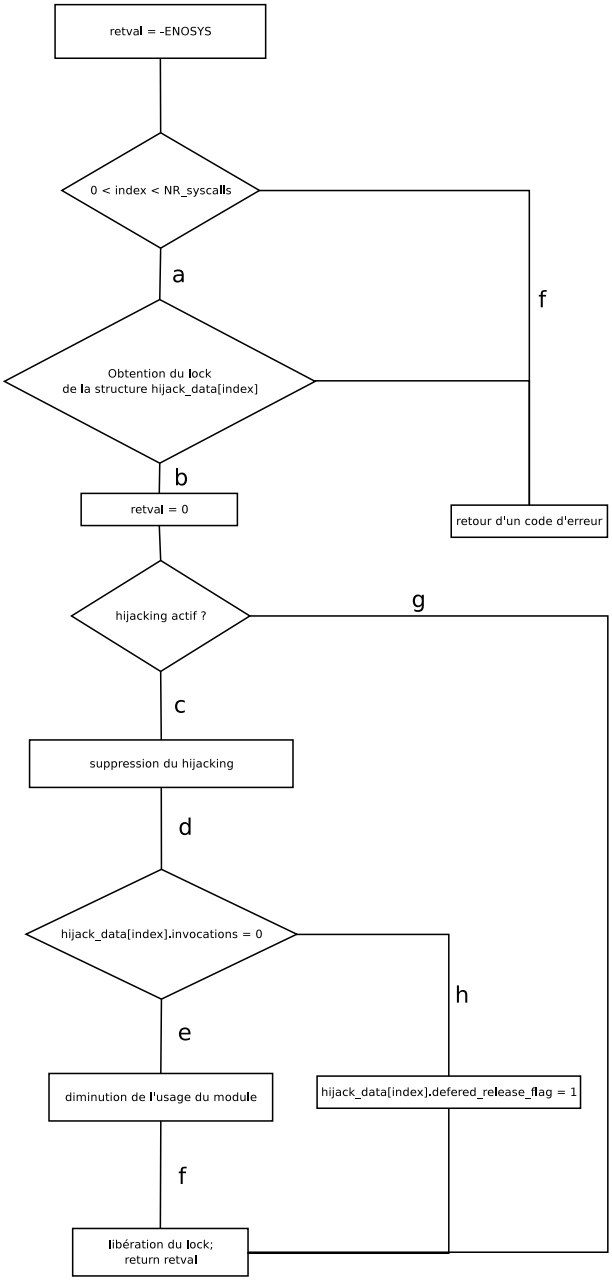

FIG. 4.7 – Algorithme de release

– h : la fonction stub est toujours utilisée, defered_release_flag est placé à 1 de telle sorte que la terminaison de la dernière invocation courante de la fonction stub entraîne une diminution de l'usage du module.

4.1.2 Les fonctions stub

Le code du kernel est ré-entrant[15]. Cela signifie qu'une même portion de code contenu dans le kernel peut être exécutée "en même temps" par plusieurs processus. Il est bien évident que, sur une machine uni-processeur, un seul processus sera actif à un moment donné mais il est tout à fait possible que d'autres processus attendent dans la même fonction que le scheduler leur donne à nouveau du temps processus (e.g. opérations I/O). Cette particularité devra être prise en compte lors de la construction du *record* NADF et nous obligera à implémenter les fonctions stub supportant plusieurs exécutions simultanées afin d'éviter des effets indésirables de "race condition".

Génération des informations au format NADF

La structure **syscall_hijack_data** possède un champ *format* qui contient un pointeur vers une chaîne de caractères représentant les informations qui doivent se retrouver dans le *record* NADF. Dans cette chaîne de caractères, chaque élément représente une information disponible et sera converti vers un élément d'un *record* NADF contenant l'instanciation de cette information. L'annexe A présente la correspondance entre les caractères et l'information contenue dans le *record* NADF. Il est important de noter qu'afin d'avoir un *record* NADF correct, il est nécessaire que l'ordre des caractères soit tel que les numéros (id du *record* NADF) correspondants soient dans un ordre strictement croissant. En effet, pour des raisons d'efficacité, le module ne vérifie pas cette contrainte et elle doit être prise en compte lors de la définition du format par l'utilisateur.

Etant donné que le code du kernel est ré-entrant, nous ne pouvons utiliser de zone de mémoire commune pour la génération des *records* NADF de l'ensemble des appels système hijackés que si nous gérons cette zone avec des mécanismes de locks et de sémaphores. Afin de réduire la complexité du code et de ne pas créer de délai trop important d'exécution du fait de l'obtention des différents locks nécessaires, nous avons décidé de ne pas utiliser de zone de mémoire commune pour construire le *record* NADF mais bien des variables locales aux différentes fonctions stub. Ceci aura pour effet d'augmenter le temps d'exécution pour l'allocation et la libération des différents buffers mais n'aura pas d'influence sur l'exécution concurrente de plusieurs appels système.

L'algorithme général que nous effectuons pour générer le *record* NADF relatif à une exécution d'un stub est le suivant :

```
max = taille de la chaîne de caractères de format
for(i=0 ; i<max ; i++)
   switch(format[i]){
         ...
      case 'x' :
         NADFid = id de x ;
         NADFvalue = valeur réel de x pour cette appel ;
         NADFsize = taille en bytes de NADFvalue ;
         ajouter les trois valeurs dans le buffer stockant le record NADF ;
```

```
            break;
        case 'y' :
            ...
    }
}
```

Comme le montre l'algorithme, les instructions nécessaires pour déterminer l'information qui est désirée sont réduites au minimum. La fonction permettant d'ajouter les trois valeurs dans le buffer stockant le *record* NADF est la fonction *addToNADFBuffer(...)* dont les spécifications se trouvent en annexe E.

Quand tous les éléments de la chaîne de caractères ont été lus, le buffer contient l'ensemble des éléments du *record* NADF à partir de son 5^{ieme} byte, il ne reste plus alors qu'à remplir les 4 premiers bytes avec la taille complète du *record* (unsigned int codé en big endian) et d'ajouter ce *record* dans le buffer du module en n'oubliant pas d'ajouter le padding nécessaire à la fin du *record* afin d'avoir un alignement sur un multiple de 4 bytes (réalisé par la fonction *addToListOfRecords(...)* dont la spécification se trouve également en annexe E. Enfin, lorsque tout le cycle est accompli, un signal est envoyé au démon pour le prévenir qu'un nouveau *record* est disponible dans le module.

Nous devons encore préciser que les opérations qui sont effectuées par le démon lui-même ne sont pas auditées si aucune opération inattendue n'a été demandée au module (ex : fin du démon suivi de la création d'un nouveau démon).

Buffering des informations

La version actuelle de notre module stocke tous les *records* qui ont été créés dans une liste chaînée. Toutes les ressources utilisées par un *record* sont libérées dès que le démon a lu l'information contenue dans ce *record*. Les ressources utilisées par des *record*s non lus au moment du retrait du module sont également libérées lors du retrait du module. Dans le cadre d'une utilisation réelle de notre solution, il est souhaitable de modifier cette stratégie car si le démon ne lit pas les informations lorsqu'elles sont disponibles, il est alors possible que notre module utilise toute la mémoire vive disponible et provoque un fort ralentissement du système hôte (voire un crash du système). Une solution possible serait l'implémentation d'une liste circulaire acceptant un nombre maximum de *records* qui remplace le *record* le plus ancien lorsqu'un nouveau *record* doit être inséré et que la liste contienne déjà le nombre maximum d'éléments. Cette solution présente toutefois le désavantage d'une perte d'informations potentielles si le démon ne lit pas suffisament vite les informations disponibles dans le module.

Génération automatique du code des stubs

Le code de la majorité des fonctions stub est identique. Les principales différences existantes sont :

- La structure **syscall_hijack_data** qui est employée dans le corps du stub.
- Le nom de l'appel système fourni dans le *record* NADF.
- Les paramètres de l'appel système.

Afin de pouvoir générer d'une façon automatique l'ensemble de ces fonctions stub, nous avons adapté un script perl initialement présent dans Syscalltrack fonctionnant de la façon suivante : l'exécution du script exploite le code commun à la majorité des fonctions stub (annexe D) ainsi que les informations relatives à chaque appel système se trouvant dans un autre fichier (annexe

??) afin de créer un fichier (syscall_hijack_autogen.c) contenant tout le code nécessaire pour les stubs. Ce fichier ne doit plus qu'être inclus dans le code du module.

Afin d'être complets, nous devons encore signaler qu'une partie des "case statement" est actuellement présente dans le script perl. Il s'agit des "case statements" qui sont dépendants des paramètres de l'appel système (eg. un appel système ayant un file descriptor comme paramètre devra exécuter des actions différentes d'un appel système ayant un pathname comme paramètre pour extraire le nom du fichier qui est accédé).

Stubs particuliers

Il existe également des fonctions stub (4 dans le cadre de ce travail : execv, exit, close, pipe) que nous n'avons pas pu générer automatiquement car leur comportement est différent de la majorité des autres appels système et que nous avons codés indépendamment dans le fichier special_hijack.c :

1. exit

   Cet appel système ne retourne jamais. Cela signifie que si nous nous contentions de la structure générale développée jusqu'à présent, nous n'aurions jamais l'information sur cet appel système (nous effectuons en effet l'appel système avant de construire le *record* NADF) et que le nombre d'invocations de cet appel système serait strictement croissant (ce qui empêcherait tout retrait futur du module).

2. execv

   Contrairement aux autres appels système qui reçoivent comme paramètre des pointeurs vers les informations à utiliser pour exécuter l'appel système, execv doit accéder aux différents registres pour obtenir ces paramètres. Ce comportement particulier ne permet donc pas de générer automatiquement cet appel système.

3. close

   L'algorithme général que nous avons développé exécute l'appel système et construit ensuite le *record* NADF. L'exécution de l'appel système original *close()* provoque la disparition de certaines structures utilisées dans la construction du *record* NADF (ex : la structure file[15]). Nous construisons donc le *record* NADF et ensuite nous exécutons l'appel système original. Toutefois, cet appel système nous pose d'autres problèmes que nous ne sommes pas parvenus à résoudre et n'est donc pas actuellement utilisable :
   - le hijacking provoque, lorsque nous terminons l'exécution de certaines applications (e.g. Mozilla), l'utilisation de références NULL. Cela se traduit par des informations dans les fichiers de logging du noyau mais le système continue à être stable.

4. pipe

   La seule particularité de cet appel système est le fait qu'il reçoit un tableau de int comme paramètre. Notre script perl n'exploitant pas correctement cette information, nous avons décidé de placer ce stub dans les stubs particuliers. L'algorithme implémenté dans ce stub est donc en tous points identique à celui implémenté dans le template.

4.1.3 Points relatifs à la non-régression du noyau

Etant donné que notre module fait partie intégrante du noyau, nous devons vérifier que l'utilisation du module ne peut pas laisser le noyau dans un état inconsistant. Nous devons donc nous assurer que les problèmes suivants sont évités :

- Le code exécuté par le module ne modifie aucun espace mémoire de façon non planifiée (e.g. dépassement de contenu d'un buffer).
- Le retrait du module laisse le noyau dans le même état que si le module n'avait jamais été inséré.
- Le code exécuté par le module n'est pas susceptible de créer des situations de deadlocks.

Une démonstration du premier point nécessiterait une étude de l'ensemble de notre code source et n'apporterait rien à la compréhension de notre implémentation. C'est pourquoi, nous ne parlerons plus du premier point. Par contre, une démonstration non formelle des deux derniers points permettra de s'assurer que l'utilisation du module ne peut pas provoquer ces types de problèmes.

Maintenir le kernel dans un état consistant

Il est impératif qu'aucun des espaces mémoires utilisés ou référencés par une fonction stub ne soient libérés avant la fin de l'exécution de cette fonction. En effet, si une structure (au niveau du module) utilisée par la fonction devait être libérée avant la fin de son exécution, nous pourrions nous retrouver dans un état inconsistant (la référence qui pointait vers la structure libérée pointe à présent vers la même zone de mémoire mais qui peut à présent contenir n'importe quoi, ou pire des informations relatives à une autre partie du kernel). Ce point est particulièrement important dans toutes les actions d'ajout et de suppression d'appels système hijackés (eg si nous désirons arrêter d'hijacker un appel système, nous devons être certains qu'aucun appel système de ce type n'est en cours avant de libérer la structure utilisée par le module pour celui-ci).

La structure **syscall_hijack_data** qui est maintenue pour chaque appel système possède un champ *invocations*. Celui-ci est un integer représentant le nombre d'invocations actuellement en cours. Au début de l'exécution d'un stub, nous incrémentons donc ce champ de 1 et à la fin de l'exécution, nous le décrémentons de 1. De cette façon, lorsque le module doit effectuer des modifications sur l'une ou l'autre structure, il peut toujours tenir compte des invocations qui sont actuellement en cours.

Un autre champ utilisé dans les fonctions stub est le champ *defered_release_flag*. Ce champ est utilisé afin de permettre la fin du hijacking d'un appel système qui avait des exécutions actives au moment où la fonction de relâchement a été invoquée et de maintenir le nombre d'utilisations du module dans un état correct (pour rappel, ce nombre doit être égal au nombre d'appels système hijackés augmenté de 1 si le démon est actuellement intéressé par les informations produites par le module). Pour bien comprendre l'importance de l'utilisation de ce champ, nous présentons un petit scénario catastrophe[5] :

- Supposons que nous n'effectuons le hijacking que d'un seul appel système. A un moment, nous décidons d'arrêter ce hijacking. Malheureusement, des invocations de cet appel système étaient en cours. Les actions provoquées par le relâchement sont alors la libération des ressources devenues superflues et le nombre d'utilisations du module est diminué de 1 (il devient égal à 0). Si nous enlevons le module à ce moment (ce qui est possible car son nombre d'utilisations est 0) alors, nous ne pouvons plus assurer que le segment de code à exécuter dans le stub, après l'exécution de l'appel système original, est valide car le retrait du module aura pour effet d'enlever la partie <stub hijack function> du schéma suivant.

[5]Cela pourrait se produire avec Snare !

```
< original system call >    <---- top of the call stack.
< stub hijack function >
< syscall dispatch function>
< user space invokes syscall>
```

L'emploi du champ *defered_release_flag* doit donc permettre de postposer la libération des ressources nécessaires pour le hijacking d'un appel système et de ne décrémenter le nombre d'utilisations du module qu'au moment où la dernière invocation de la fonction stub se termine.

L'utilisation combinée des deux champs précédents permet d'être certains que le scénario catastrophe que nous avons présenté ne peut pas se produire dans notre implémentation car en tenant compte des spécifications de la figure 4.6, nous obtenons :

1. Si aucune invocation n'est en cours, alors la valeur du champ invocation est 0 et l'usage du module est diminué de 1.
2. Si au moins une invocation est en cours, alors la valeur du champ invocation est supérieur à 1 (la première instruction que nous exécutons dans une fonction stub est l'augmentation du nombre d'invocations de cette fonction stub). Le fait de demander le *release* de l'appel système provoquera le replacement de la référence vers l'appel système original dans la **syscall_table** (les prochaines invocations de cet appel système ne provoqueront plus une exécution de la fonction stub) et la valeur du champ *defered_release_flag* sera placée à 1. Le module ne peut donc pas être retiré tant qu'il y a une invocation active de la fonction stub puisque l'usage du module ne sera diminué de 1 qu'au moment de la terminaison de la dernière invocation actuellement active. Nous insistons également sur le fait que cette diminution aura lieu car les invocations de l'appel système postérieures au release ne provoquent plus l'exécution du code de la fonction stub mais bien directement l'appel système original.

Le champ *orig_syscall* maintient une référence vers l'appel système original. Ceci permet d'invoquer son exécution pendant l'exécution du stub et également de replacer l'appel système original dans la **syscall_table** lorsque le hijacking prend fin.

S'assurer qu'il n'est pas possible d'avoir des deadlocks

L'accès à certaines structures nécessite l'utilisation de locks et de sémaphores. Il est important de se rendre compte que l'utilisation de ceux-ci ne peut pas provoquer des situations de deadlocks qui figeraient certaines fonctionnalités du noyau. Etant donné que les parties de notre module ne détiennent jamais qu'un maximum de un et un seul lock (sémaphore), nous démontrons d'une façon intuitive que le code présent, entre l'obtention du lock (sémaphore) et le relâchement de celui-ci, ne peut se bloquer indéfiniment :

1. Utilisation du sémaphore de la structure **syscall_hijack_data**.
   (a) Utilisation dans la fonction stub
      i. Au début de l'exécution de cette fonction, le sémaphore est utilisé dans la section de code suivante :

```
down(&hdata->sem);
orig_syscall_func = (stub_func_2)hdata->orig_syscall;
actualFormat = hdata->format;
up(&hdata->sem);
```

Les deux instructions qui sont exécutées pendant la détention du sémaphore sont des assignations de pointeurs et ne sont pas bloquantes.

ii. Une seconde et dernière utilisation du sémaphore est réalisée dans la portion de code suivante :

```
down(&hdata->sem);
if (atomic_read(&hdata->invocations) == 1 && hdata->defered_release_flag == 1)
kfree(actualFormat);
MOD_DEC_USE_COUNT;
}
up(&hdata->sem);
```

Les instructions qui sont exécutées sont deux tests, une assignation, la libération d'une ressource devenue inutile ainsi que la diminution du nombre d'usages du module. Ici encore, aucune des instructions n'est bloquante.

(b) Utilisation lors du hijacking : *hijack_ syscall(...)*

Le sémaphore doit être obtenu au début de l'éxécution de la fonction *hijack_ syscall(...)* et est libéré à la fin de l'exécution de la fonction. Toutes les instructions[6] qui sont exécutées ne sont pas bloquantes et si le résultat d'une de celles-ci rend impossible l'exécution de la suite, alors, cette suite n'est pas exécutée mais la portion de code suivante est exécutée avant de terminer la fonction :

```
hdata->defered_release_flag = defered_flag;
up(&hdata->sem);
return retval;
```

(c) Utilisation lors du release : *release_ syscall(...)*

Le même principe que le point précédent est applicable.

2. Utilisation du lock de la liste chaînée de *records*

Comme nous l'avons déjà précisé, notre module maintient les *records* qui n'ont pas encore été lus par le démon dans une liste chaînée. L'ajout et le retrait d'éléments de cette liste chaînée ne peut s'effectuer qu'en obtenant un lock.

(a) Ajout d'un élément : *addToListOfRecords(...)*

La construction du *record* se fait sans l'utilisation de locks et l'ajout de celui-ci dans la liste chaînée se fait en exécutant la portion de code suivante :

```
down(&masax_lock);
if(tail != NULL){
  tail->next = newRecord;
}
tail = newRecord;
if(head == NULL){
  head = newRecord;
}
up(&masax_lock);
```

[6]Le nombre d'instructions étant important, nous renvoyons à l'annexe ?? pour consulter le code source.

Nous pouvons constater qu'aucune de ces instructions n'est bloquante.

(b) Retrait d'un élément : *auditmodule_ read(...)*
Le lock doit être obtenu au début de l'exécution de la méthode et est relâché à la fin de celle-ci. Toutes les instructions[7] qui sont exécutées ne sont pas bloquantes et si le résultat d'une de celles-ci rend impossible l'exécution de la suite, alors le lock est relâché et le code d'erreur correspondant est retourné.

4.1.4 Communication avec le démon

A présent que le module est fonctionnel, il nous reste à réaliser la communication avec un processus s'exécutant en "user-space" et responsable de collecter les informations disponibles dans le module. Nous avons attribué cette responsabilité à un démon. Nous avons besoin de trois mécanismes de communication distincts : un premier pour effectuer les communications dans le sens démon-module (pour effectuer la configuration du module) et un second pour transmettre les *records* du module vers le démon et un troisième qui doit permettre au module de prévenir le démon quand de l'information devient disponible. Les deux premiers moyens de communication sont possibles grâce au fait que le module crée un fichier dans le système de fichier /proc (le nom du fichier est masax) et le dernier est inspiré de snare.

Configuration du module

Le module peut être configuré par le démon en utilisant la fonction *ioctl(...)* qui permet à un module d'offrir un point d'entrée pour l'exécution de commande spécifique[16]. La signature de la fonction est *int ioctl(int d, int request, ...)* où *d* représente un file descriptor valide, *request* représente la commande qui doit être exécutée et ... représente un argument facultatif dont la taille maximum est de 4 bytes (ce qui est suffisant pour donner la valeur d'un pointeur vers une structure contenant des informations). Dans le cadre de notre travail, le *d* correspond donc au fichier que le démon a ouvert pour recevoir les informations (cf infra) alors que *request* peut prendre les valeurs définies dans le fichier syscall_hijack_public.h (voir F) :

- HIJACK (10) afin d'effectuer le hijacking d'au moins un appel système
- RELEASE (11) afin d'arrêter le hijacking d'un appel système
- RELEASE_ALL (12) afin d'arrêter le hijacking de tous les appels système hijackés

Lorsque le *request* est HIJACK, le dernier paramètre correspond à un pointeur vers le premier élément d'une liste chaînée contenant une ou des structures de type :

```
struct hijack_information{
  int syscall_index;
  char* format_string;
  struct hijack_information* next;
};
```

Le champ *next* doit être égal à NULL pour le dernier élément de cette liste. Quand le *request* est RELEASE, le dernier paramètre contient l'index dans la **syscall_table** de l'appel système dont il faut arrêter le hijacking et si le *request* est RELEASE_ALL, alors le dernier paramètre n'est pas exploité.

[7] Le nombre d'instructions étant important, nous renvoyons à l'annexe ?? pour consulter le code source.

Prévenir le démon qu'une information est disponible

Notre module doit être capable de prévenir le démon quand un nouveau *record* NADF est disponible. Ce mécanisme est nécessaire afin d'éviter une situation de *buzy waiting* où le démon lirait continuellement sur la sortie du module en attendant un nouveau *record* NADF. Pour réaliser cette fonctionnalité, notre module tient une référence vers la structure représentant le processus du démon (*task_struct*). Cette dernière permet d'envoyer un signal (SIGIO) au démon lorsqu'un nouveau *record* est disponible. La référence vers le démon est obtenue au moment où le démon ouvre le fichier /proc créé par le module.

Transmettre l'information disponible au démon

A présent que nous pouvons configurer le module et prévenir le démon de la présence de *record* NADF, il ne nous reste plus qu'à implémenter un mécanisme permettant de transmettre ce *record*. Du côté du démon, tout se passe comme s'il accédait à un fichier classique et il lui suffit de faire une opération de lecture sur le fichier /proc/masax qui a été créé par le module. Le module, quant à lui, implémente l'ensemble des opérations autorisées sur le fichier. Une opération de lecture se résume donc aux événements suivants :

1. Le démon crée un buffer pour contenir le *record* NADF et invoque la fonction read.
2. Le kernel informe le "driver du fichier" (ie : notre module) qu'une opération de lecture est nécessaire.
3. Le module a accès au buffer du démon et y place le *record* (ensuite, le module libère les ressources utilisées dans le kernel pour ce *record*).
4. L'opération de lecture est terminée et le démon a le *record* dans son buffer.

Actuellement, nous avons également fait l'hypothèse simplificatrice que le nombre de bytes du buffer du démon est toujours au moins égal au nombre de bytes composant le *record* NADF à transmettre.

4.1.5 Quelques aspects de sécurité relatifs au module

Comme nous travaillons dans un contexte de sécurité, nous avons voulu fournir certaines caractéristiques afin de le rendre plus robuste en cas d'attaque. Les différentes attaques que nous proposons supposent que l'utilisateur malveillant a réussi à obtenir le statut de *super user*.

Le premier type d'attaques possibles est le fait qu'un utilisateur malveillant pourrait remplacer le démon par un nouveau programme effectuant les mêmes opérations et exécutant certaines autres manipulations (pour rappel, les opérations du démon ne sont initialement pas auditées). Le scénario de l'attaque serait alors de supprimer le démon original et de lancer un nouveau processus qui se connecterait sur notre module. La réaction que nous avons prévue pour ce type d'attaque est le fait que, si le démon se détache de notre module et qu'un nouveau processus se rattache au module, alors, ce processus devient un processus normal pour notre module et toutes les actions exécutées par celui-ci sont auditées. Malheureusement, il suffirait que ce pseudo-démon effectue un filtre et retire les *records* le concernant pour que la situation semble "normale". Il faudrait cependant :

- Obtenir les privilèges de *super-user*.
- Que les clients ne remarquent pas la substitution.

Un autre type d'attaque possible serait d'effectuer le retrait du module. Un mécanisme simple de défense est d'augmenter l'usage du module au moment de son chargement. De cette façon, son nombre d'utilisations est toujours au moins égal à 1 et il devient impossible de le retirer du kernel. Nous n'avons toutefois pas utilisé ce mécanisme dans le cadre de notre mémoire car nous voulions toujours pouvoir le retirer.

Un troisième type d'attaque serait d'essayer d'intercepter, ou d'avoir accès, aux informations disponibles depuis le module. La solution que nous avons implémentée ne permet qu'à un seul processus ayant les privilèges de "super-user" de se connecter sur la sortie du module.

4.1.6 Possibilités d'extension

Le module que nous avons développé constitue une excellente base pour un système futur plus complet :

- S'il est nécessaire d'obtenir de nouvelles informations, il suffit d'ajouter un nouveau case statement dans le template des fonctions stub (ou de l'ajouter dans le script perl s'il s'agit d'une information propre à un paramètre de certains appels système), d'exécuter à nouveau le script perl (sans oublier de modifier également le fichier special_hijack.c) et de recompiler le module.
- Si une fonction stub devient plus spécifique, il suffit de ne plus la générer automatiquement et d'introduire son code dans le fichier special_hijack.c.

D'autres extensions sont encore possibles mais nécessiteraient une quantité de travail plus importante :

- Si nous désirons effectuer un filtre sur certains utilisateurs, il faudrait prévoir un champ dans la structure **syscall_hijack_data** permettant de stocker l'information sur le id intéressant, fournir également cette information lorsque le module est configuré et enfin l'exploiter dans la fonction stub. Mais ici encore, le fait d'avoir utilisé un script pour générer le code des fonctions stub permettra de ne coder qu'une seule fois la logique et de la retrouver dans l'ensemble des fonctions stub (sauf pour les appels système spéciaux).
- Notre module a été développé en utilisant un kernel de type 2.4.x. Il serait intéressant de le rendre également compatible avec des kernel de type 2.2.x et 2.6.x

4.2 Implémentation d'un démon

4.2.1 Introduction

Dans cette section, nous allons aborder plus spécifiquement l'implémentation et la réalisation du démon **nadfd**. Les responsabilités de ce dernier sont au nombre de deux :

- Récupérer les *records* NADF qui ont été construits par notre module et les transmettre à des applications clientes.
- Permettre la configuration du module (i.e. définir les appels système devant être interceptés et le contenu des *records* NADF relatifs à ces appels système).

Comme nous l'avons déjà signalé, l'information d'audit des appels système est mise à disposition via le fichier spécial /proc/masax par le module **syscall_hijack**. Pour des raisons de sécurité, il est important qu'une seule application ait accès à l'information disponible sur le fichier /proc/masax. En effet si une deuxième application possédait aussi la capacité de lire les enregistrements NADF sur le fichier spécial, la première application n'aurait pas l'ensemble des enregistrements étant donné que ceux-ci ont déjà été retirés par la deuxième application. Il

faut donc s'assurer que tous les enregistrements réalisés soient bien mis à la disposition d'une seule application à la fois.

Mais nous cherchons aussi à mettre à la disposition de plusieurs applications en même temps les enregistrements NADF du Noyau audité. Et cela sans qu'il y ait de perte d'enregistrements pour les clients. Par exemple, nous voulons une application qui enregistre l'ensemble des *records* dans un fichier et une autre qui fait l'analyse en temps réel avec l'outil ASAX.

4.2.2 Implémentation du démon

Notre démon **nadfd** se compose de deux parties distinctes :
- Une première partie se connecte au fichier /proc/masax pour communiquer avec le module d'audit d'appel système, ainsi le démon peut envoyer la configuration des appels système à auditer.
- La deuxième partie permet à différents clients de se rattacher au démon et de recevoir les *records* NADF reçus du module.

Communication avec le module syscall_hijack

La communication entre le démon **nadfd** et le module d'audit se fait via un fichier particulier /proc/masax. Dans un sens le démon envoie la configuration au module via la fonction **ioctl** [8]. Ensuite, le démon lit sur le fichier /proc/masax les enregistrements NADF pour les transmettre aux clients comme pour un fichier normal.

On utilise le signal SIGHUP pour informer le démon qu'il doit relire les fichiers de configuration des appels système à hijacker. Cette méthode permet de réinitialiser en temps réel notre configuration des appels système à auditer sans devoir arrêter le démon. Il faut effectuer la commande "kill -HUP $(pidof nadfd)" ou en exécutant le script **sendHUP** présent sur le cd (répertoire nadfd).

Communication avec les clients d'enregistrements NADF

Afin de réaliser le rôle de serveur de *record* au format NADF, la connection des clients au démon se fait via des sockets UNIX [18].

L'utilisation des sockets présente plusieurs avantages :
- Les sockets permettent une communication entre deux processus distincts.
- Le client et le serveur ne doivent pas se trouver obligatoirement sur la même machine.
- La connexion peut être initialisée orientée connexion ou orientée sans connexion. Si elle est orientée connexion, nous sommes certains que le client a bien reçu l'enregistrement NADF. Encore une fois pour des raisons de sécurité il est important de s'assurer que l'ensemble des enregistrements NADF soit délivré aux clients .

Pour l'instant, le démon est uniquement configuré en réseau local et donc seuls les clients se trouvant sur la machine peuvent se connecter au démon. Mais une simple modification de paramétrage d'initialisation des sockets permet la connexion au sein d'un réseau.

Notre choix d'utilisation des sockets a été principalement motivé par cette possibilité d'extension vers un serveur au sein d'un réseau. L'idée étant le développement futur d'une machine spécialisée au sein d'un réseau dans le stockage et dans l'analyse des données. Une machine

[8] voir 4.1.4

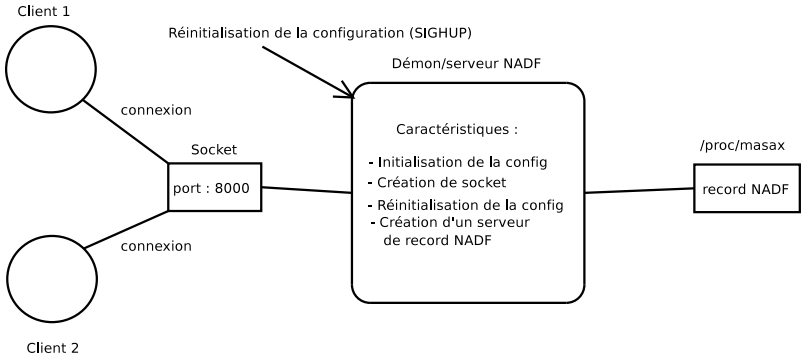

FIG. 4.8 – Réalisation d'un serveur et connexion au device /proc/masax

sera donc spécifiquement dédiée à la récupération des enregistrements NADF d'un réseau d'ordinateurs.

4.2.3 Réalisation d'un premier client

Dans un premier temps, nous avons réalisé un client se connectant au démon afin de recevoir les *records* NADF et de stocker ceux-ci dans un fichier pouvant être utilisé directement par ASAX sans aucun adapteur de format (i.e. avant de commencer l'enregistrement des *records* NADF, le *header* des fichiers NADF est enregistré dans le fichier). Ce client écrase le fichier d'audit précédent à chaque fois qu'il est réexécuté (/tmp/nadf_log).

4.2.4 Réalisation d'un client d'analyse temps réel

La réalisation d'un client ASAX n'a pas été effectuée dans le cadre de ce mémoire, mais nous tenions à présenter ici les grandes lignes à suivre pour le développement d'un client de ce type.

Il est possible d'étendre ASAX en fournissant de nouvelles routines écrites en C. On peut, ainsi, générer un nouvel exécutable en liant de nouvelles routines C ou I/O routines au noyau ASAX. Cette création d'un nouvel exécutable se fait à l'aide de la commande masax.

$$masax - c - fa < I/Oroutine > nom_du_nouveau_asax$$

Notre format étant déjà au format NADF, il n'est pas nécessaire de réaliser un *format adapter*. Mais nous allons utiliser les fonctionnalités du *format adaptater* afin de chercher les informations sur le fichier spécial /proc/audit à l'aide de la fonction ioctl.

La réalisation d'un *format adaptater* afin de le lier avec le noyau ASAX se fait à partir de code C. Il est nécessaire d'écrire trois routines :

- vopen() exécute les actions nécessaires pour construire et préparer le fichier virtuel d'audit à la lecture.
- vread() lit l'enregistrement (*record*) suivant et le transforme si nécessaire au format NADF.
- vclose() ferme le fichier virtuel, précédemment ouvert par vopen().

```
<Liste_event> : := <Numero_event> “ :” <Nom_event>
                 “ :” <Description> “ :” <Liste_class>”;”
<Numero_event> : := [0-9]+
<Nom_event> : := [a-z]*
<Description> : := [A-Z][a-z]+
<Liste_class> : := Class
<Class> : := [A-Z][A-Z]
<Liste_class> : := <Liste_class> “,” <Class>
```

FIG. 4.9 – Définition de la grammaire pour la configuration des classes

L'idée est alors de réaliser l'implémentation de ces trois routines et d'étendre ASAX. La routine vopen() se charge simplement d'établir la connexion sur le démon, la routine vclose() implémentera la fermeture de la connexion avec le démon et la routine vread() se chargera d'implémenter la requête de demande de l'enregistrement NADF suivant.

4.2.5 Implémentation d'un mécanisme de configuration

Le mécanisme de la configuration est également un point important de notre système, car il fournit aux utilisateurs de notre application un outil à la fois pratique et facilement configurable. Il faut aussi noter qu'un bon mécanisme de configuration est nécessaire, étant donné qu'un système d'audit sur le noyau Linux coûte en performance mais aussi en espace de stockage.

Il est donc important de limiter l'audit des appels système au strict nécessaire, mais également de limiter les informations nécessaires. Le mécanisme de configuration permet à l'utilisateur de configurer les classes d'appels système qu'il souhaite auditer mais aussi de personnaliser l'information de sortie au format NADF.

Pour la réalisation de notre configuration, nous nous sommes basés principalement sur le système de configuration de BSM dans Solaris. Notre système de configuration se compose de deux fichiers, le fichier /etc/security/class et le fichier /etc/security/control.

BSM réalise des audits sur des classes et pas directement des appels système. Nous avons repris ce principe. L'idée de ce type de configuration est de créer un ensemble de classes qui regroupe de manière logique des appels système. Par exemple, nous pouvons regrouper tous les appels système liés au fichier (open, creat, close ...) dans une classe, ou encore les appels système liés à l'administration (reboot, chroot,...). Ce mode de configuration se justifie par le fait que, en règle générale, l'administrateur cherche à auditer un ensemble logique d'appels système.

Le fichier de configuration /etc/security/control

Ce premier fichier définit les classes. Chaque appel système est contenu dans une ou plusieurs classes. Une règle relative à un appel système est composée par les éléments suivants :

- Un numéro de règle
- Le nom de l'appel système
- Une courte description
- Une liste de classes : une classe étant définie par deux lettres majuscules.

```
01 :fork :Fork :EC ;
02 :open :Open :FC ;
03 :creat :Creat :FC ;
04 :execve :Execv :EC ;
05 :mkdir :Mkdir :FC ;
06 :unlink :Unlink :FC ;
07 :chown :Chown :FX ;
08 :lchown :Lchown :FX ;
09 :chmod :Chmod :FX ;
10 :symlink :Symlink :FC ;
11 :link :Link :FC ;
12 :rename :Rename :FC ;
13 :reboot :Reboot :AC ;
14 :truncate :Truncate :FC ;
15 :chroot :Chroot :AC ;
16 :setuid16 :Setuid :EC ;
17 :setuid :Setuid :EC ;
18 :setreuid :Setuid :EC ;
19 :setgid :Setgid :EC ;
20 :setregid :Setregid :EC ;
21 :mknod :Mknod :FC ;
22 :chown16 :Chown :FX ;
23 :setresuid :Set :EC ;
24 :setresgid :Set :EC ;
25 :setreuid16 :Set :EC ;
26 :setgid16 :Set :EC ;
27 :setregid16 :Set :EC ;
%28 :select :Select :AC ;
29 :dup :Dup :GH ;
30 :getpid :Get :GH ;
31 :getuid :Get :GH ;
32 :umask :Uma :GH ;
```

FIG. 4.10 – Exemple de configuration du fichier /etc/security/class

```
<Set_classe> : := <Classe>
<Set_classe> : := <Set_classe> “;” <Classe>
<Classe> : := [A-Z][A-Z]
<Classe> : := [A-Z][A-Z] “ :” <Liste_Output>
<Liste_Output> : := <Out>
<Out> :== u|g|e|f|c|n|P|M|U|G|O|N|L
<Liste_Output> : := <Liste_Output> “,” <Out>
```

FIG. 4.11 – Définition de la grammaire pour la configuration des classes à auditer

```
AC :u,g,e,f,P,n,c ;
FC :u,g,e,f,P,n,c ;
FX :u,g,e,f,P,n,c ;
EC :u,g,e,f,P,n,c ;
GH :u,g,e,f,P,n,c ;
```

FIG. 4.12 – Exemple de configuration du fichier /etc/security/control

Le fichier de configuration /etc/security/control

Le deuxième fichier définit les classes à auditer. Il est aussi possible de personnaliser les informations disponibles dans le *record* NADF. Une règle est composée des éléments suivants :

- Le nom d'une classe (défini dans le fichier /etc/security/class)
- Une suite de lettres définissant l'information de sortie de l'enregistrement NADF (voir exemple 4.10).

Nous avons vu ci-avant que nous utilisions la fonction ioctl pour envoyer la configuration des appels systèmes à auditer. A l'aide de cette fonction, nous transmettons un pointeur vers le premier élément d'une liste chaînée. Si la liste est vide, nous ne transmettons rien au module étant donné qu'il est inutile d'envoyer une liste vide. Il faut encore noter que le dernier élément de la liste doit pointer vers NULL (cf).

4.13

Pour réaliser le parseur, nous avons utilisé les outils **bison**[9] et **flex** [10], deux logiciels du projet GNU. Le premier est un générateur de parseur et le deuxième est un générateur d'analyseur lexical. Ils génèrent du code C à partir de la définition d'un ensemble de *Token* pour **flex** et d'une grammaire LALR pour **bison**.

[9]La version de bison utilisée est 1.87a

[10]la version de flex utilisée est 2.5.31

```
typedef struct hijack_information{
        int syscall_index ;
        char* format_string ;
        struct hijack_information *next ;
 }Hijack ;
```

FIG. 4.13 – Structure de la liste des éléments à hijacker

Nom	Description	ID	char du format[11]
UID	ID de l'utilisateur	2	u
GID	ID du groupe	3	g
EUID	ID réel de l'utilisateur	4	e
EGID	ID réel du groupe	5	f
PID	ID du processus	21	M
COMM	commande utilisée pour exécuter le processus	22	U

TAB. 4.1 – Informations auditées au niveau du process

L'utilisation de ces deux outils nous a permis dans un premier temps de réaliser un parseur propre et facilement modifiable. Mais il est vrai que l'utilisation de **flex** et de **bison** demande un certain temps d'apprentissage. Ces deux outils s'avèrent pratiques et facilement configurables. Le développement de nouvelles fonctionnalités dans le système de configuration se fera simplement en modifiant les fichiers .lex et .y de nos parseurs.

Nous avons réalisé dans un premier temps un parseur pour le fichier /etc/security/class qui génère une liste chaînée contenant la définition de chaque classe avec les appels système le constituant.

Nous avons ensuite réalisé un deuxième parseur qui génère la liste des appels système à auditer avec le format de sortie spécifique. Le format de sortie sont les lettres "n,u,g,e,f,c,P,M,U,G,O,N,L,G,a,b,F,d,l" chaque lettre représentant une information. Nous trouverons la correspondance entre les lettres et l'information sortie du *record* NADF dans la figure 4.1 et la figure 4.2. Les lettres n'ayant pas de correspondance sont laissées pour une extension future du module **syscall_hijack**. Il peut arriver que deux ou plusieurs appels système sont définis dans plusieurs classes n'ayant pas le même format de sortie. Si nous exécutons un audit sur ces classes en même temps, alors,le format de sortie de l'appel système est une fusion des deux

format de sortie défini dans les deux classes.

4.3 Efficacité de l'implémentation

4.3.1 Tests des performances de la machine hôte

Unixbench

1. Description des benchmarks réalisés :

   (a) Le benchmark "System call overhead" consiste à déterminer le nombre d'itérations d'une boucle contenant les appels systèmes dup(), close(), getpid(), getuid() et umask() qu'il est possible d'effectuer en une minute.

   (b) Le benchmark "Spawn" consiste à créer un processus fils qui se termine immédiatement après avoir effectué son propre fork.

   (c) Le benchmark "Execl" consiste à exécuter un processus qui change à chaque fois "d'incarnation".

   (d) Le benchmark "Shell" consiste à réaliser une série d'opérations qu'une application réelle utilise. Il est basé sur un script shell qui effectue un tri alphabétique d'un fichier dans un nouveau fichier, détermine le "dump" octal du résultat et place un tri

Nom	Description	ID	char du format	Remarques
NAME	Nom de l'appel système	1	n	
RETURNVALUE	Valeur de retour de l'appel système original	6	c	Quand utilisé dans *fork*, correspond au *pid* du processus enfant
FILENAME	Chemin du fichier accédé	20	P	Utilisable quand *fd*
ERRORCODE	Code d'erreur	23	G	Seulement pour *exit*
ADDR	adresse d'une zone mémoire	50	Z	paramètre *addr* de certains appels système
OLDLEN	taille d'une zone mémoire	51	a	paramètre *old_len* de certains appels système
NEWLEN	nouvelle taille d'une zone mémoire	52	b	paramètre *new_len* de certains appels système
FLAGS	*flags* utilisés	53	F	paramètre *flags* de certains appels système
NEWADDR	nouvelle adresse d'une zone mémoire	54	d	paramètre *new_addr* de certains appels système
LEN	taille	55	l	paramètre len de certains appels système

TAB. 4.2 – Informations auditées au niveau de l'appel système

numérique dans un troisième fichier. Le programme "grep" est exécuté sur le fichier trié alphabétiquement et place le résultat dans un fichier sur lequel le nombre de mots le composant (wc) est déterminé et placé dans un dernier fichier. Enfin, tous les fichiers créés sont effacés. Ce test est effectué en exécutant 1 processus ainsi qu'en exécutant 8 et 16 processus concurrents.

2. Conditions dans lesquelles les différents benchmarks ont été réalisés :

   (a) Tous les benchmarks ont été réalisés sur une machine équipée d'un processeur Athlon cadencé à 1.2 Ghz et disposant de 512 MB de RAM. Le système d'exploitation installé sur la machine est Debian GNU/Linux et la version du kernel est le 2.4.21.

   (b) Les benchmarks ont été réalisés dans les conditions suivantes :

      i. Afin de pouvoir déterminer les caractéristiques de base du système hôte, nous avons effectué une première série de benchmarks sur un système ne contenant aucun élément de notre implémentation.

      ii. Les benchmarks ont ensuite été réalisés sur un système exécutant tous les éléments de notre implémentation mais sans qu'aucun appel système ne soit hijacké. Les résultats que nous avons obtenus pour cette partie étant similaires à ceux obtenus au point précédent, nous n'en parlerons pas dans la partie résultat.

      iii. Les benchmarks ont été réalisés dans un troisième temps en effectuant le hijacking des appels système utilisés intensivement par les différents benchmarks mais sans communication avec le démon (afin de déterminer le coût engendré par le module) :
         - Pour le premier, les appels système *dup()*, *getpid()*, *getuid()* et umask() ont été hijackés.
         - Pour les deux benchmarks suivants, les appels système *fork()*, *exit()* et *execve()* ont été hijackés.
         - Les appels système considérés comme les plus critiques par les implémenteurs de SNARE (27) ont été hijackés pour le dernier benchmark.

      iv. Enfin, nous avons effectué un test dans les mêmes conditions que le précédent mais en incluant un démon actif (ie : qui traite les informations disponibles au niveau du module) et un client opérationnel enregistrant tous les *records* NADF dans un fichier.

3. Résultats et interprétations des différents benchmarks[12]

   (a) System call overhead

      i. Le nombre moyen d'itérations dans la boucle effectuées par seconde est de 784585. Le temps imparti au processus pour effectuer le test a été utilisé à concurrence de plus ou moins 50% en user space et à concurrence de plus ou moins 50% en kernel space.

[12] Un récapitulatif de ces résultats se trouve dans le tableau 4.3

ii. L'insertion du module et le hijacking des différents appels système font chuter les performances à 126537 exécutions de la boucle par seconde. Ceci représente donc des performances atteignant un peu plus de 16% du système non modifié. Cette perte importante de performance s'explique par le fait que les appels système qui sont hijackés sont des appels système qui sont exécutés rapidement dans leur version originale. Le fait de construire un *record* NADF pour ceux-ci représente donc un coût important par rapport à la fonction originale. Plus précisément, l'implémentation originale d'un appel système tel que *getuid()* ne fait qu'aller chercher la valeur du champ d'une structure et la retourner. Nous réalisons alors exactement la même opération pour obtenir l'information à ajouter dans le *record* NADF. Il est donc logique que le coût pour effectuer le hijacking d'appel système simple soit aussi important. L'analyse du temps passé en user space et en kernel space confirme cette hypothèse : le temps imparti au processus pour effectuer le benchmark a été utilisé à concurrence de plus ou moins 7.5% en user space et environ 90% en kernel space.

iii. Le fait d'activer le démon et d'insérer un client permet d'obtenir un nombre moyen d'itérations de 67847. Ceci représente 8.6% des performances du système original et 53% de celles du module sans utilisation du démon. Cette nouvelle baisse de performance s'explique par le fait que le processus démon a utilisé de manière intensive le CPU (jusque 40% du temps CPU) provoquant une disponibilité moins grande de celui-ci pour le benchmark. Le temps imparti au processus pour effectuer le test a été utilisé à concurrence de plus ou moins 5% en user space et à concurrence d'un peu plus de 50% en kernel space. Nous remarquons donc que le benchmark n'a pas eu de temps CPU pendant environ 45% de son temps d'exécution : ceci explique donc la chute de performance de 53 % par rapport au module seul car si le benchmark avait disposé du même temps CPU, nous aurions des performances similaires à celles obtenues au point précédent.

(b) Spawn

i. Le nombre moyen d'itérations dans la boucle effectuées par seconde est de 7418. Le temps imparti au processus pour effectuer le test a été utilisé à concurrence de plus ou moins 9% en user space et à concurrence de plus ou moins 83% en kernel space.

ii. L'insertion du module et le hijacking des différents appels système n'a pas d'influence sur les performances du système et la répartition du temps passé respectivement en user space et en kernel space est similaire au point précédent. Nous pouvons donc remarquer que le hijacking et la création de *record* NADF n'a aucun impact significatif sur l'exécution d'un appel système devant originellement exécuter un grand nombre d'actions. Cette constatation est logique puisque les actions que nous exécutons pour construire le *record* NADF sont proportionnellement moins importantes (voire négligeables) pour ce type d'appel système.

iii. Une fois que le module est inséré et que le hijacking des appels système utilisés est effectué, nous obtenons un nombre moyen d'itérations de 4896. Ceci représente 66% des performances du système original. Le temps imparti au

Test	Base		Module		Module + démon	
	Moyenne	Ecart Type	Moyenne	Ecart Type	Moyenne	Ecart Type
syscall overhead	784585	1313	126537	265	67847	971
Spawn	7418	1270	7427	1165	4896	669
Execl	2481	490	2372	123	2326	248
Shell (1 process)	1100	1.5	1106	0.6	1060	1.7
Shell (8 process)	160	0	163	0	156	0.4
Shell (16 process)	81	0	82	0	78	0.4

TAB. 4.3 – Résultat des *benchmarks* unixbench

processus pour effectuer le test a été utilisé à concurrence de plus ou moins 8% en user space et à concurrence d'un peu plus de 60% en kernel space. Cette chute des performances est encore une fois provoquée par le fait que le démon et le client ont consommé des ressources CPU pendant le temps d'exécution du benchmark. Nous pouvons remarquer que si nous prenons en compte uniquement le temps CPU dont a disposé le benchmark, et non plus le temps total alloué au benchmark, que nous obtenons alors des performances pour cette dernière exécution du benchmark de l'ordre de 89% de celles du système original.

(c) Execl

i. Le nombre moyen "d'incarnations" par seconde est de 2481. Le temps imparti au processus pour effectuer le test a été utilisé à concurrence de plus ou moins 41% en user space et à concurrence de plus ou moins 56% en kernel space.

ii. De la même façon qu'au point précédent, l'insertion du module et le hijacking des différents appels système n'a pas d'influence sur les performances du système ainsi que sur la répartition du temps passé respectivement en user space et en kernel space.

iii. Une fois que le module est inséré et que le hijacking des appels système utilisés est effectué, nous obtenons un nombre moyen d'incarnations de 2326. Ceci représente 93% des performances du système original. Le temps imparti au processus pour effectuer le test a été utilisé à concurrence de plus ou moins 41% en user space et à concurrence d'un peu plus de 56% en kernel space.

(d) Shell

i. Le nombre moyen d'exécutions par minute est de 1100 pour l'exécution avec 1 processus. Les exécutions concurrentes ont des performances de 160 exécutions par minute (8 processus) et 81 exécutions par minute (16 processus).

ii. L'insertion du module et le hijacking ne provoquent pas de dégradation des performances du système.

iii. L'activation du démon et du client fait passer les performances à respectivement 1060, 156 et 78 exécutions par minute. Ceci correspond à des performances de l'ordre de 96-97 % du système original.

lmbench

1. Description des benchmarks réalisés
   (a) Le benchmark "Simple syscall" : est le temps pour écrire un byte à /dev/null, nous testons ici l'appel système write().
   (b) Le benchmark "Simple read" : est le temps pour lire un fichier de 64K0, nous testons ici l'appel système read().
   (c) Le benchmark "Simple write" est le temps nécessaire pour écrire dans un fichier test.
   (d) Le benchmark "Simple stat" est le temps nécessaire pour exécuter un appel système stat (status d'un fichier) sur un fichier test.
   (e) Le benchmark "Simple fstat" est le temps nécessaire pour exécuter un appel système stat sur le file descriptor test.
   (f) Le benchmark "Simple open/close" est le temps nécessaire pour exécuter l'ouverture et le fermeture sur un fichier test.
   (g) Le benchmark "Select of X fd's" est le temps nécessaire pour exécuter l'appel système select X fois sur un file descriptor test.
   (h) Le benchmark "Process fork+exit" est le temps nécessaire pour exécuter l'appel système fork (création d'un processus fils) et exit.
   (i) Le benchmark "Process fork+execve" est le temps nécessaire pour créer un processus fils et d'exécuter un programme
   (j) Le benchmark "Process fork + /bin/sh -c" est le temps nécessaire pour créer un processus fils et d'exécuter le programme sh.
2. Conditions dans lesquelles les différents benchmarks ont été réalisés
   (a) Nous avons réalisé les benchmarks sur une machine équipée d'un processeur Athlon cadencé à 1.8 Mhz et disposant de 256 MB de RAM. Le système d'exploitation installé sur la machine est Debian GNU/Linux et la version du kernel est le 2.4.25.
   (b) Les benchmarks ont été réalisés dans les conditions suivantes :
      i. Dans un premier temps, nous avons réalisé une première série de dix tests sans aucune modification du système original. Ce premier test nous servira de référence pour les trois autres. (Etat 1)
      ii. Dans un deuxième temps, nous avons réalisé une série de dix tests avec l'ajout du module n'interceptant aucun appel système. (Etat 2)
      iii. Dans un troisième temps une autre série de dix tests a été réalisée avec ajout du module en auditant les appels système considérés comme les plus critiques de snare, avec le démon en activité ainsi que le client. (Etat 3)
      iv. Enfin, nous avons effectué une quatrième série de dix tests avec l'ajout du module en auditant les appels système considérés comme les plus critiques de snare, avec le démon et le client désactivé. (Etat 4)
3. Résultats et interprétations des différents benchmarks

Les résultats obtenus se trouvent dans le tableau 4.4.

	Etat 1	Etat 2	Etat 3	Etat 4	
Simple syscall :	0.16	0.16	0.16	0.16	microsecondes
Simple read :	0.38	0.35	3.47	2.08	microsecondes
Simple write :	0.32	0.3	3.48	2.52	microsecondes
Simple stat :	1.03	1.02	0.98	1.01	microsecondes
Simple fstat :	0.4	0.38	0.43	0.45	microsecondes
Simple open/close :	1.81	1.66	77.77	4.05	microsecondes
Select on 10 fd's :	1.91	1.73	6. 98	2.86	microsecondes
Select on 100 fd's :	15.46	14.6	45.24	22.67	microsecondes
Select on 250 fd's :	24.9	21.51	76.94	49.28	microsecondes
Select on 500 fd's :	51.43	48.45	100.96	74.09	microsecondes
Process fork+exit :	107.64	107.41	192.25	130.7	microsecondes
Process fork+execve :	358.92	360.41	897.85	500.27	microsecondes
Process fork+/bin/sh -c :	2132.9	2116.27	8048.3	2428.46	microsecondes

TAB. 4.4 – Moyenne des dix résultats de lmbench par état.

- Premièrement, on ne constate aucune différence entre l'état 1 et l'état 2. Il ne coûte donc rien, en terme de performance, au système d'insérer le module si aucun appel système n'est audité.
- On constate comme dans Unixbench, une chute des performances avec l'insertion et le hijacking des appels système (état 4). On constate en général une augmentation de 2 à 4 microsecondes (état 4) dans l'exécution d'un appel simple système (read, write) sur un temps d'exécution normal de 0.30. Cela représente un temps d'éxécution 10 fois plus important.
- On peut justifier les mauvaises performances dans l'état 3, obtenues avec les benchmarks Simple open/close, Process fork+execve et Process fork + /usr/bin/sh -c, par le fait que le démon et le client ont consommé des ressources CPU pendant le temps d'exécution du benchmark. Le temps nécessaire pour exécuter l'appel système est dès lors beaucoup plus long. Les résultats repris à l'état 4 sont plus proches de la réalité concernant le coût réel de l'hijacking d'un appel système étant donné que l'on a désactivé le démon et le client.
- Une dernière remarque concerne l'exécution d'appel système plus lourd comme le benchmark Process fork +/bin/sh -c : on peut constater que le coût de l'audit (2428,46 microsecondes) ne représente qu'une faible augmentation en comparaison avec le coût normal d'exécution qui est de 2132.9 microsecondes.

Utilisation normale de la machine

Les mesures que nous avons effectuées jusqu'à présent étaient des mesures objectives des performances de notre système. Nous avons également réalisé un test plus subjectif qui consistait à utiliser nos machines en laissant s'exécuter notre implémentation et en effectuant le hijacking des appels système considérés comme les plus critiques par les implémenteurs de snare (27). Nos impressions sont que l'utilisateur ne perçoit pas de dégradations des performances du système lors d'une utilisation classique (utilisation d'une suite bureautique, surf, utilisation du mail, programmation, ...).

Conclusion du point de vue des performances

Le test le plus défavorable pour notre module est sans aucun doute le test relatif au "system call overhead". Toutefois, les résultats de ce test trouvant une explication logique, nous pouvons conclure que notre réalisation est efficace puisque nous arrivons à des performances de l'ordre de 96% pour les benchmarks simulant des activités intensives susceptibles d'être réalisées par des programmes réels. De plus, nous n'avons pas eu d'impression de baisse de performances lors de l'utilisation journalière de nos systèmes exécutant le module et le démon.

4.3.2 Simulation de quelques attaques

Afin d'utiliser notre application d'une façon pratique et réaliste, nous avons dans un premier temps élaboré deux scénarii d'attaques possibles : le premier consiste à accéder à un fichier particulier après avoir tenté d'exécuter un programme particulier et le second à effectuer plusieurs tentatives de *login*. Le but de ces deux premières attaques était principalement de se rendre compte des points forts et des points faibles de notre solution dans le cas concret d'une détection d'intrusion afin d'en tirer des enseignements et d'orienter les directions futures que pourraient prendre ce travail. Etant donné que les résultats que nous avons obtenus ne nous donnaient pas entière satisfaction, nous avons décidé, a partir de nos premières constations, de faire évoluer[13] notre logiciel afin de permettre une détection plus fine, et avons testé cette "nouvelle" version avec deux attaques plus réalistes : tentative d'exécution de **su** (avec utilisation d'un mauvais mot de passe) et une attaque exploitant une faille présente sur certains noyaux de l'appel système *mremap*.

Détection d'accès à des fichiers sensibles

La première attaque que nous avons simulée est celle d'un utilisateur tentant de devenir *root* en utilisant la commande **su** et qui effectue par la suite une copie du fichier /etc/passwd. Nous avons configuré notre module afin qu'il intercepte les appels système pouvant être interceptés par SNARE et que les *records* NADF contiennent toutes les informations que notre implémentation permettait[14] d'obtenir.

Les règles RUSSEL que nous avons utilisées pour détecter l'attaque sont présentées à la figure 4.14.

L'attaque que nous avons réalisée a bien été détectée lors de l'analyse du fichier d'audit généré par notre application.

Détection de login

La seconde attaque que nous avons simulée est celle d'un utilisateur qui essaie d'ouvrir une session sur un terminal en utilisant un nom d'utilisateur correct mais en essayant plusieurs mots de passe incorrects. Nous avons configuré notre module de la même façon que lors du test précédent et avons utilisé les règles RUSSEL présentes à la figure 4.15.

L'attaque a une nouvelle fois été détectée lors de l'analyse de l'*audit trail*. Malheureusement, une analyse plus approfondie des données disponibles montre que nous sommes susceptibles de générer un grand nombre d'alarmes négatives et nous invite à tirer nos premiers enseignements concernant une utilisation pratique de notre application.

[13] Il s'agit simplement d'ajouter la possibilité d'auditer plus d'informations

[14] A ce moment, nous ne détections pas encore le code d'erreur qui est le paramètre de l'appel système *exit*.

```
init_action;
begin
        println('==============> Starting init action rule');
        trigger off for_next find_exec
end;

rule find_exec;
begin
        if NAME %= 'execve' and FILENAME %= '/bin/su'
                --> begin                                                    10
                        println('su vu');
                        trigger off for_next find_passwd(UID)
                    end
        fi;
        trigger off for_next find_exec
end;

rule find_passwd(uid : string);
begin
                                                                             20
        if
                UID %= uid and NAME %= 'open' and FILENAME %= '/etc/passwd'
                --> begin
                        println('acces passwd !');
                        println('UID est : ', strToInt(UID))
                    end;

                true
                --> trigger off for_next find_passwd(uid)
        fi                                                                   30

end.
```

FIG. 4.14 – Règles RUSSEL pour la détection de fichiers sensibles

```
init_action;
begin
        println('==============> Starting init action rule');
        trigger off for_next find_exec
end;

rule find_exec;
begin
        if NAME %= 'execve' and FILENAME %= '/bin/login'
                --> begin                                                    10
                        println('login vu');
                        println('UID est : ', strToInt(PID));
                        trigger off for_next find_login(UID)
                    end
        fi;
        trigger off for_next find_exec
end;

rule find_login(uid : string);
begin                                                                        20

        if
                UID %= uid and NAME %= 'execve' and FILENAME %= '/bin/login'
                --> begin
                        println('second login vu ! !');
                        println('UID est : ', strToInt(PID))
                    end;

                true
                --> trigger off for_next find_login(uid)                     30
        fi

end.
```

FIG. 4.15 – Règles RUSSEL pour la détection de login

Les premiers enseignements des simulations précédentes

Comme nous avons pu le constater, les informations produites par le module sont exploitables pour la détection des attaques que nous avons simulées.

Mais ces différentes simulations ont également permis de se rendre compte des faiblesses de notre système. Dans le cas de l'accès aux fichiers sensibles, la première chose que nous détections était l'exécution du programme **su**. Mais si nous pouvons détecter l'exécution du programme, nous sommes incapables de déterminer si celle-ci s'est terminée en donnant les droits de *super user* à l'utilisateur l'ayant invoqué ou en gardant les mêmes droits qu'avant l'invocation (si le mot de passe était incorrect par exemple). Ceci est dû au fait que l'exécution de l'appel système *execve* (qui est un des appels système utilisé pour le lancement de programmes) est identique pour toutes les exécutions possibles d'un programme (i.e. pour l'appel système, le fait que **su** donne ou non l'accès aux privilèges de *super-user* est considéré comme une exécution normale).

Dans le cas de la tentative de login, le même genre de problème fait son apparition : si nous pouvons trouver les exécutions du programme "login", nous ne pouvons pas dire s'il s'est terminé par la création d'une session ou par un refus. De même, nous ne pouvons pas fournir d'informations relatives à l'identité de l'utilisateur car le programme login est exécuté en tant que root !

Détection de tentatives de su

Afin de contrôler que notre implémentation est capable de fournir des informations permettant de détecter des événements plus précis que ceux que nous avons détectés jusqu'à présent, nous proposons des règles permettant de détecter le fait qu'un utilisateur effectue des tentatives infructeuses d'exécution du programme **su** afin de devenir *super user*. Etant donné les premiers enseignements que nous avons tirés des premières attaques, nous avons élaboré les règles en effectuant le raisonnement suivant : puisque nous ne pouvons pas détecter, grâce aux informations disponibles dans les *records* de l'appel système *execve,* si le programme s'est terminé en accordant les droits de *super user* ou non, nous allons exploiter le code d'erreur présent dans le paramètre de l'appel système *exit* afin de déterminer le "type" de terminaison. Nous avons donc étendu notre implémentation afin de capturer les nouveaux paramètres qui nous semblaient intéressants et avons utilisé les règles RUSSEL présentent à la figure4.16.

L'analyse de l'*audit trail* a permis cette fois-ci de détecter toutes les tentatives non réussies d'exécution de **su** que nous avons faites et a également permis d'obtenir l'identité de la personne exécutant l'attaque. La figure 4.17 contient la sortie de l'exécution d'ASAX sur un *audit trail* contenant deux tentatives de **su**.

Détection de l'attaque mremap

Cette attaque consiste à exécuter un programme C permettant d'obtenir un shell ayant les droits de *super user*[15]. Les noyaux vulnérables à cette attaque sont tous les noyaux inférieurs à 2.2.25, 2.4.24 et 2.6.2 qui n'ont pas été *patchés*. Nous avons réalisé cette attaque sur un noyau 2.4.21 non *patché* et sommes effectivement parvenus à obtenir un shell avec des droits de *super user*. Afin de pouvoir détecter cette attaque, nous avons audité les appels système

[15]Une explication détaillée de cette attaque sortant du cadre de ce travail, nous renvoyons le lecteur intéressé par les détails à `http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt` .

```
init_action;
begin
        println('==============> Starting init action rule');
        trigger off for_next find_exec
end;

rule find_exec;
begin
        if NAME %= 'execve' and FILENAME %= '/bin/su'
                --> begin                                                    10
                        trigger off for_next find_exit_su(PID)
                    end
        fi;
        trigger off for_next find_exec
end;

rule find_exit_su(pid : string);
begin

        if                                                                   20
                PID %= pid and NAME %= 'exit' and COMM %= 'su'
                --> begin
                        println('exit of su detected -> wrong password !!');
                        println('PID est : ',strToInt(PID));
                        println('UID est : ',strToInt(UID));
                        println('error_code est : ', strToInt(ERRORCODE))
                    end;

                true
                --> trigger off for_next find_exit_su(pid)                   30
        fi

end.
```

FIG. 4.16 – Règles RUSSEL pour la détection de tentatives de su

FIG. 4.17 – Sortie d'une exécution d'ASAX avec un *audit trail* contenant deux tentatives de su

relatifs à la gestion de la mémoire (*mmap2*, *oldmmap*, *vfork*, *mremap*, *munmap*) et Xavier Martin a élaboré les règles RUSSEL de la figure 4.18 qui permettent, comme le montre la figure 4.19, de générer une alerte lorsque cette attaque est exécuté et de connaître l'identité de l'utilisateur ayant lancé l'attaque.

4.4 Comparaison avec les logiciels précédemment analysés

4.4.1 Syscalltrack

Notre module a initialement été créé en se basant sur l'implémentation de Syscalltrack. Le fait d'avoir analysé et ensuite "nettoyé" le code de cette application nous a permis d'obtenir un code source plus compréhensible : nous avons en effet supprimé les structures et mécanismes qui ne nous étaient pas nécessaires (actions effectuées avant l'exécution de l'appel système original, après l'exécution de l'appel système original, ...). Ceci a également pour effet que notre logiciel est plus facilement extensible (cf annexe C) : si nous désirons ajouter de nouvelles informations dans les *records* NADF, il suffit de modifier le *template* des fonctions stub, de les regénerer automatiquement en utilisant le script perl, et de prendre en compte les nouveaux paramètres dans les fichiers de configuration.

Mais la simplification du code a eu pour effet que nous avons perdu des fonctionnalités par rapport à Syscalltrack : il n'est, par exemple, plus possible d'auditer uniquement un utilisateur particulier. Ceci pourrait constituer un point négatif de notre implémentation mais il ne faut pas perdre de vue que l'application que nous avons développée est dédiée à ASAX et que c'est ce dernier qui prend en charge ce type de fonctionnalités.

Le but premier de Syscalltrack étant d'effectuer du debugging, le logiciel n'effectue pas le *buffering* des informations auditées : si un processus exploite la sortie de Syscalltrack, alors les informations seront utilisées mais si aucun processus n'exploite cette sortie, alors toutes les informations seront perdues. Ceci peut être dramatique dans le cadre de la détection d'intrusion et notre module implémente ce mécanisme de *buffering*.

Nous pouvons donc affirmer que l'utilisation commune d'ASAX et de notre implémentation permet d'obtenir pratiquement[16] toutes les fonctionnalités proposées par Syscalltrack tout en offrant des possibilités d'extensions simplifiées (lors de la phase de tests, l'ajout de nouvelles informations dans les records NADF a été réalisé dans un temps inférieur à 1 heure).

4.4.2 SNARE

SNARE permet le *hijacking* de 27 appels système alors que notre solution permet théoriquement[17] le *hijacking* de tous les appels système. Notre implémentation permet également de sélectionner les informations relatives à chaque appel système que nous désirons retrouver dans l'*audit trail* alors que SNARE ne propose qu'un format fixe.

Le fait de créer un audit au format NADF nous permet d'étendre également beaucoup plus facilement notre logiciel : peu importe la nouvelle information que nous voulons inclure dans un *record*, il suffit de connaître son *id*, sa valeur et sa taille pour pouvoir effectivement l'intégrer au *record*. En effet, SNARE utilise également des structures pour stocker les informations d'audit (certaines structures correspondant à un certain groupe d'appels système) et si nous

[16] La seule fonctionnalité que nous ne pouvons pas offrir est la non-exécution d'un appel système.

[17] Nous insistons sur le terme théorique puisque nous n'avons pas eu l'occasion de tester l'ensemble des appels système.

```
init_action;
begin
        trigger off for_next no_more_vma
end;

rule no_more_vma;
var count : integer;
begin
   if (match('old_mmap', NAME) = 1) and (strToInt(RETURNVALUE) = -12) /* code d'erreur : plus
      --> trigger off for_next mremap(strToInt(PID))                                    10
   fi;
   trigger off for_next no_more_vma
end;

rule mremap(pid :integer);
begin
   if (strToInt(PID) = pid)
      --> if
            (match('exit', NAME) = 1)
               --> begin                                                                20
                   end;
            (match('mremap', NAME) = 1) and (strToInt(RETURNVALUE) > 0)
               --> trigger off for_current report;

            true
               --> trigger off for_next mremap(pid)
         fi;
      true --> trigger off for_next mremap(pid)
   fi
end;                                                                                    30

rule report;
begin
   println('ATTENTION :');
   println('----------');
   println('Tentative d exploitation de la vulnerabilite des appels systemes mremap et munmap');
   println('URL : http ://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt');
   println('CVE : CAN-2004-0077');
   println;
   println('UID : ', strToInt(UID));                                                     40
   println('EUID : ', strToInt(EUID));
   println
end.
```

FIG. 4.18 – Règles RUSSEL pour la détection de tentatives de mremap

```
Opening nadf_log
Processing audit trail ...
ATTENTION :
---------
Tentative d exploitation de la vulnerabilite des appels systemes mremap et munmap
URL : http ://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt
CVE : CAN-2004-0077

UID :  1000
EUID : 1000


end of audit trail reached.
Processing completion rules ...

End of emulation.
```

FIG. 4.19 – Sortie d'une exécution d'ASAX avec un *audit trail* contenant une attaque mremap

désirons ajouter une nouvelle information, il est nécessaire de modifier ces structures ainsi que toutes les parties de code utilisant ces structures.

Il y a certains points pour lesquels SNARE est supérieur à notre implémentation :

- Communication des *records* entre le module et le démon : SNARE permet de communiquer un *record* en plusieurs étapes si le buffer du démon est trop petit alors que notre application fait l'hypothèse que le buffer du démon sera toujours suffisament grand. Il s'agit toutefois d'un problème pouvant être facilement résolue.
- Filtre des événements audités : SNARE permet, par exemple, d'auditer uniquement les informations relatives à un certain utilisateur. Mais l'analyse du code source montre que tous les appels système sélectionnés sont en fait audités et que le filtre est appliqué dans le démon. Donc, même si un filtre est seléctionné par l'utilisateur, SNARE audite autant d'appels système que notre implémentation.

Enfin, il y a certains points où notre implémentation est nettement supérieure :

- Génération automatique de code des fonctions stub
- Impossibilité de retirer notre module et de laisser le noyau dans un état inconsistant.
- Comme il n'est pas possible d'auditer n'importe quel appel système avec SNARE, il n'est pas possible pour ce dernier de détecter la dernière attaque que nous avons réalisé.

Ici aussi, nous pouvons donc affirmer que notre solution est supérieure et permet de plus, d'exploiter directement la puissance d'ASAX pour analyser l'*audit trail*.

Conclusion et travaux futurs

La politique que nous avons choisi d'appliquer au début de notre travail (réutilisation de code existant) nous a permis d'obtenir une solution finale complète. Mais, afin d'évaluer si nous avons atteint les objectifs que nous nous étions fixés dans le premier chapitre, nous proposons de les parcourir à nouveau et de les confronter au résultat que nous avons obtenu.

Le premier objectif était de présenter certaines des possibilités d'implémentations permettant une détection d'intrusion sur GNU/Linux. Nous l'avons réalisé tout au long de notre second chapitre.

La présentation de certains logiciels actuels pertinents pour la détection d'intrusion constituait notre deuxième objectif. Celui-ci était l'objet de notre troisième chapitre qui a permis de faire connaissance avec les logiciels que nous avons estimés être les plus pertinents.

Notre objectif suivant était la réalisation d'un logiciel de détection d'intrusion pour GNU/Linux ayant les particularités suivantes :

- Etre dédié à ASAX et donc offrir directement une sortie au format NADF.
- Etre facilement configurable afin de permettre l'audit des informations jugées intéressantes par l'administrateur de la machine.
- Etre facilement modifiable et adaptable afin de pouvoir offrir rapidement une solution à des besoins futurs.

Nous pouvons affirmer que la solution que nous proposons rencontre ces différents points :

- Les *records* construits par le module sont au format NADF.
- Les fichiers de configuration permettent de déterminer facilement les appels système à auditer ainsi que les informations devant être présentes dans les *records* NADF.
- La phase de simulation d'attaques nous a assuré que notre implémentation était facilement adaptable puisque la possibilité de générer automatiquement les fonctions stubs nous a permis d'ajouter rapidement les différentes informations qui n'étaient pas encore présentes dans les *records* NADF.

Notre quatrième et dernier objectif a également été atteint puisque nous sommes arrivés au terme de notre implémentation et avons eu la possibilité de conduire quelques tests dont les résultats montrent que notre implémentation a un coût raisonnable du point de vue des performances du système et qu'il est possible de détecter les intrusions que nous avons proposées.

Le résultat général de notre travail est donc largement positif puisque nous avons obtenu un système utilisable prouvant la faisabilité d'un IDS performant pour GNU/Linux spécialement dédié à ASAX. Mais il existe encore de nombreuses pistes qui peuvent être explorées afin d'obtenir une solution plus complète, parmi lesquelles nous retrouvons :

- Le module n'a été réalisé que pour les noyaux de la série 2.4.X. Il serait intéressant d'effectuer les modifications afin qu'il puisse être déployé sur les noyaux 2.2 (qui sont encore utilisés sur les serveurs) et 2.6.
- Améliorer le module :

  - afin d'avoir un mécanisme de *buffering* limitant l'espace pouvant être utilisé par les *records* n'ayant pas encore été lus par le démon.
  - afin de ne plus devoir effectuer la communication entre le module et le démon en utilisant uniquement des *records* entiers
  - afin de capturer plus d'informations
  - afin d'optimiser la construction des *records* NADF.
  - afin d'offrir la possibilité d'auditer les appels système uniquement pour certains utilisateurs, pour l'accès à certains fichiers, ...
- Générer automatiquement les fichiers de configuration en fonction des règles RUSSEL qui seront utilisées pour analyser l'*audit trail*.
- Réaliser un protocole de communication entre les clients et le démon.
- Etendre le démon pour la réalisation d'un serveur nadfd dans un réseau d'ordinateurs (analyse du coût de transfert des records dans le réseau).
- Se conformer plus fidèlement aux standards CC et C2.
- Elaborer un set de règles RUSSEL pertinentes pour la détection d'intrusion sous GNU/Linux.

Bibliographie

[1] Asax advanced sequential file analysis product, novembre 1993.

[2] Intersect Alliance. Comprehensive event log monitoring, Mars 2003.

[3] Sergio Noberto Gutiérrez Beltran. Program monitoring based on asax. Master's thesis, Université Catholique de Louvain, 2003.

[4] K. Levitt B.Mukherhee, L. Heberlein. Network intrusion detection. *IEEE Network*, 1994.

[5] Rohnert Hans Peter Sommerlad Stal Michael Buschman Franck, Meunier Regine. *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley and Sons Ltd, 1996.

[6] Common Criteria. Common criteria version 2.1. common language to express common needs. http ://csrc.nist.gov/cc/CC-v2.1.html, 1999.

[7] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[8] K. Jain and R. Sekar. User-level infrastructure for system call interposition : A platform for intrusion detection and confinement. pages 19–34.

[9] William Arbaugh Jesus Molina. Using independant auditors as intrusion detection systems. Technical report, Department of Computer Science, University of Maryland.

[10] et al. Kurt Wall. *Linux Programming Unleashed*. SAMS, 2001.

[11] Zhen Liu. A lightweitght intrusion detection system for the cluster environment. Master's thesis, Mississippi State University, 2003.

[12] Sun Microsystems. *Sunshield Basic Security Module*. 1997.

[13] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, 1997.

[14] Departement of defense standard. *Departement of defense trusted computer system evaluation criteria*. 1985.

[15] Daniel P.Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

[16] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly, 2001.

[17] A. Somayaji. Operating system stability and security through process homeostasis. doctoral dissertation, 2002. University of New Mexico.

[18] W.Richard Stevens. *Unix Network programming*. Prentice Hall Software series, 1990.

[19] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[20] David A. Wheeler. Secure programming for linux and unix howto. http ://www.dwheeler.com/secure-programs/, Mars 2003.

Annexe A

Liste des informations auditées :

Cette annexe présente les informations qu'il est actuellement possible d'obtenir en utilisant le module. Le nom correspond à l'information obtenue, ID correspond à l'id qui sera utilisé dans le record NADF et la colonne "char du format" correspond au caractère qui doit être présent dans la chaîne de caractères définissant les informations contenues dans le record NADF.

A.1 Process

Nom	Description	ID	char du format[1]
UID	ID de l'utilisateur	2	u
GID	ID du groupe	3	g
EUID	ID réel de l'utilisateur	4	e
EGID	ID réel du groupe	5	f
PID	ID du processus	21	M
COMM	Commande utilisée pour exécuter le processus	22	U

A.2 Appels système

Nom	Description	ID	char du format	Remarques
NAME	Nom de l'appel système	1	n	
RETURNVALUE	Valeur de retour de l'appel système original	6	c	Quand utilisé dans *fork*, correspond au *pid* du processus enfant
FILENAME	Chemin du fichier accédé	20	P	Utilisable quand *fd*
ERRORCODE	Code d'erreur	23	G	Seulement pour *exit*
ADDR	adresse d'une zone mémoire	50	Z	paramètre *addr* de certains appels système
OLDLEN	taille d'une zone mémoire	51	a	paramètre *old_len* de certains appels système
NEWLEN	nouvelle taille d'une zone mémoire	52	b	paramètre *new_len* de certains appels système
FLAGS	*flags* utilisés	53	F	paramètre *flags* de certains appels système
NEWADDR	nouvelle adresse d'une zone mémoire	54	d	paramètre *new_addr* de certains appels système
LEN	taille	55	l	paramètre len de certains appels système

Annexe B

Structure des sources de l'application

B.1 Module

Le code du module est réparti dans les répertoires hijack et perl.

B.1.1 Répertoire hijack

- syscall_hijack.c : partie principale du module.
- special_hijack.c : code des fonctions *stub* ne pouvant pas être générées en utilisant le script perl.
- syscall_hijack_private.h, syscall_hijack_public.h : fichiers *header*.
- createAutogen : script démarrant la génération de la majorité des fonctions *stub*.

B.1.2 Répertoire perl

- gen_syscalls.pl : script perl pour la génération de la majorité des fonctions *stub*.
- syscalls.dat : fichier contenant la signature des appels système (utilisé par gen_syscalls.pl).
- templates/syscall_hijack_stub.tmpl.Linux : template de la fonction *stub* (utilisé par gen_syscalls.pl).

B.2 démon

Le code du démon est réparti dans le répertoire nadfd.

B.2.1 répertoire nadfd

- Control.lex : code pour le générateur lexical flex pour le fichier de configuration /etc/security/control.
- Control.y : code pour le générateur de parseur bison, pour le fichier de configuration /etc/security/control.
- class.lex : code pour le générateur lexical flex pour le fichier de configuration /etc/security/class.
- class.y : code pour le générateur de parseur bison, pour le fichier de configuration /etc/security/class.
- nadfd.c : contient le code pour la réalisation du démon/serveur (prérequis : Control et Class .lex et .y).
- list.c : contient le code de la génération des listes utilisées dans le projet.

- list.h : fichier *header* .
- utils.h : contient le tableau de correspondance nom appel système/*integer*.
- audit : est un script à placer dans /etc/init.d, il insère le module (modification du *path* nécessaire) et initialise le démon **nadfd**.
- sendHUP : est un script qui envoie un signal QUiT au démon **nadfd.**
- class : exemple de fichier de configuration class.
- control : exemple de fichier de configuration control.
- README : contient les informations nécessaire à la mise en place de la partie démon **nadfd** et client.

B.3 Le client

- client.c : contient le code du client qui se charge de logger les *records* NADF dans le fichier nadf_log.

Annexe C

Mode d'emploi de l'application

C.1 Déploiement et utilisation

1. Le déploiement de l'application se fait en deux étapes :
   - Compilation du module (attention les sources de votre noyau sont nécessaires et le compilateur utilisé doit être le même que celui utilisé lors de la compilation du noyau) avec le commande *make* dans le répertoire hijack. Vous obtenez ainsi un module appelé **syscall_hijack.o.** Pour installer le module exécutez la commande *"make install"*.
   - Compilation du démon et du client : exécutez la commande *make* dans le répertoire nadfd. Vous obtenez deux exécutables **nadfd** et **client**. Utilisez les fichiers /etc/security/class et /etc/security/control pour configurer l'application (des informations supplémentaires concernant la configuration du démon se trouvent dans le fichier README). Utilisez "make install" (en *super user*) pour copier les deux exécutables dans le répertoire /usr/local/bin et le script d'initialisation d'audit dans le répertoire /etc/init.d/.
2. Pour initialiser l'audit du système :
   - il faut exécuter en *super user* le script "/etc/init.d/audit start".
   - Si vous souhaitez initialiser le système d'audit au démarrage du système d'exploitation utilisez la commande "ln -s /etc/init.d/audit /etc/rc2.d/S99audit".
   - les *records* NADF sont contenus dans le fichier /tmp/nadf_log et ne sont donc pas conservé à l'arrêt de la machine.
3. Pour arrêter l'audit du système
   - exécutez le script "/etc/audit.d/audit stop"
4. Il est aussi possible de réinitialiser la configuration "à chaud". Pour cela, modifiez les fichiers de configuration (class et control dans le répertoire /etc/security) et lancez le script sendHUP.

C.2 Comment ajouter de nouvelles informations dans les records NADF

C.2.1 Informations communes à tous les appels système

Pour obtenir des informations qui sont présentes pour l'ensemble des appels système (e.g. informations contenues dans **current**,...), il suffit d'ajouter un *case* dans le *switch-case* du template syscall_hijack_stub.tmpl.Linux et de régénérer les fonctions *stub* en utilisant le script. Si cette information doit également être auditée pour les appels système qui ne sont pas générés automatiquement, il ne faut pas oublier d'adapter le fichier special_hijack.c.

C.2.2 Informations propres à certains appels système

Ce point concerne l'obtention d'informations qui sont disponibles en fonction des paramètres de l'appel système. Il faut alors éditer le script perl et modifier la fonction createConditionalCode en ajoutant le code relatif à ce paramètre et régénérer les fonctions *stub* en utilisant le script. Si cette information doit également être auditée pour les appels système qui ne sont pas générés automatiquement, il ne faut pas oublier d'adapter le fichier special_hijack.c.

C.2.3 Au niveau du démon.

Pour ajouter une nouvelle lettre correspondant à une nouvelle information à hijacker, il faut modifier les fichiers control.lex et control.y. Dans le premier fichier, il suffit d'ajouter la nouvelle lettre, et dans le second fichier, il faut augmenter la taille de la macro "sizeformat" de 1 et ajouter la lettre dans la fonction *BuildFormat()* (attention de respecter la position en tenant compte de l'id qui sera utilisé dans le record NADF pour cet élément).

Annexe D

Template des fonctions stub

Cette annexe présente le template que nous avons utilisé pour générer la majorité des fonctions stub de notre module. Il correspond donc au fichier qui doit être adapté si nous désirons modifier le comportement des fonctions stub ou si nous désirons permettre le *logging* de nouvelles informations.

```
/* stub function for syscall 'SYSCALL NAME' (SYSCALL ID). */
asmlinkage static int
stub_syscall_%%SYSCALL_ID%%(%%SYSCALL_PARAMS_RECV%%) {
        unsigned int i = 0;
        unsigned int max = 0;
        unsigned short NADFid = 0;
        unsigned short NADFsize = 0;
        unsigned long bufferSize = BASEBUFFERSIZE;
        unsigned int tempInt = 0;
        char* actualFormat = NULL;                                          10
        void* NADFvalue = NULL;
        unsigned long NADFoffset = 4;
        void* base = kmalloc(bufferSize, GFP_KERNEL);
        typedef int
                (*stub_func_%%SYSCALL_ID%%)(%%SYSCALL_PARAMS_RECV%%);

        stub_func_%%SYSCALL_ID%% orig_syscall_func;


        int orig_syscall_retval = 0; /* the original syscall retval */      20

        struct syscall_hijack_data* hdata = &hijack_data[%%SYSCALL_ID%%];

        /* ok, an invocation is in progress */
        atomic_inc(&hdata->invocations);
```

```
down(&hdata->sem) ;

orig_syscall_func = (stub_func_%%SYSCALL_ID%%)hdata->orig_syscall ;          30
actualFormat = hdata->format ;

up(&hdata->sem) ;


/* invoke the original syscall*/
orig_syscall_retval = (*orig_syscall_func)(%%SYSCALL_PARAMS_NAMES%%) ;


if (base == NULL){                                                           40 if
        printk(KERN_NOTICE "impossible d'allouer espace dans stub\n");
    goto done;
}

if ((current == masaxdaemon_task_struct || masaxdaemon_task_struct == NULL) if
     && !BAD_MANIPULATION_DETECTED){
        kfree(base);
        goto done;
}
                                                                             50
max = strlen(actualFormat);
//
//INV : soit 0<n<i<max
 //     alors tout format[n] a été utilisé pour créer le record NADF
//VAR : max-i >0
for(i = 0; i<max; i++){                                                      for
        switch(actualFormat[i]){
                case 'n' : // syscall's name
                        NADFid = 1 ;
                        NADFvalue = "%%SYSCALL_NAME%%" ;                     60
                        NADFsize = strlen(NADFvalue) ;

                        base = addToNADFBuffer(base,&bufferSize, &NADFoffset,
                              htons(NADFid), htons(NADFsize), NADFvalue) ;
                        if(base == NULL){
                               goto done ;
                        }
                        break ;
                case 'u' : //user id
                        NADFid = 2 ;                                         70
                        tempInt = htonl(current->uid) ;
                        NADFvalue = &tempInt ;
                        NADFsize = sizeof(unsigned int) ;
```

```
        base = addToNADFBuffer(base,&bufferSize, &NADFoffset,
              htons(NADFid), htons(NADFsize), NADFvalue);
        if(base == NULL){
               goto done;
        }
        break;
case 'g' : //group id
        NADFid = 3;
        tempInt = htonl(current->gid);
        NADFvalue = &tempInt;
        NADFsize = sizeof(unsigned int);

        base = addToNADFBuffer(base,&bufferSize, &NADFoffset,
              htons(NADFid), htons(NADFsize), NADFvalue);
        if(base == NULL){
               goto done;
        }
        break;
case 'e' : //effective user id
        NADFid = 4;
        tempInt = htonl(current->euid);
        NADFvalue = &tempInt;
        NADFsize = sizeof(unsigned int);

        base = addToNADFBuffer(base,&bufferSize, &NADFoffset,
              htons(NADFid), htons(NADFsize), NADFvalue);
        if(base == NULL){
               goto done;
        }
        break;
case 'f' : //effective group id
        NADFid = 5;
        tempInt = htonl(current->egid);
        NADFvalue = &tempInt;
        NADFsize = sizeof(unsigned int);

        base = addToNADFBuffer(base,&bufferSize, &NADFoffset,
              htons(NADFid), htons(NADFsize), NADFvalue);
        if(base == NULL){
               goto done;
        }
        break;
case 'c' : //return value of original syscall
        NADFid = 6;
        tempInt = htonl(orig_syscall_retval);
        NADFvalue = &tempInt;
        NADFsize = sizeof(int);
```

```
                        base = addToNADFBuffer(base,&bufferSize, &NADFoffset,
                              htons(NADFid), htons(NADFsize), NADFvalue);
                        if(base == NULL){
                              goto done;
                        }
                        break;
                  case 'M' : //pid of current process
                        NADFid = 21;                                          130
                        tempInt = htonl(current->pid);
                        NADFvalue = &tempInt;
                        NADFsize = sizeof(unsigned int);
                        base = addToNADFBuffer(base,&bufferSize, &NADFoffset,
                              htons(NADFid), htons(NADFsize), NADFvalue);
                        if(base == NULL){
                              goto done;
                        }
                        break;
                                                                              140
                  case 'U' : //command to start the process
                        NADFid = 22;
                        NADFsize = strnlen(current->comm,MAXCOMMAND);
                        NADFvalue = kmalloc(NADFsize, GFP_KERNEL);
                        if (NADFvalue == NULL){
                           break;
                        }
                        strncpy(NADFvalue,current->comm, NADFsize);
                        base = addToNADFBuffer(base,&bufferSize, &NADFoffset,
                              htons(NADFid), htons(NADFsize), NADFvalue);     150
                        if(base == NULL){
                              goto done;
                        }
                        break;

                  %%CONDITIONAL_CASE_CODE%%
            }
      }
      *(unsigned long *)base=htonl(NADFoffset);
      addToListOfRecords(base);                                               160
      kfree(base);

      signal_auditd();

done :
      down(&hdata->sem);

      if (atomic_read(&hdata->invocations) == 1 && hdata->defered_release_flag == 1) {
```

```
        /* we couldn't reduce the module's use count when the syscall */
        /* was originally released (as marked by the 'defered' flag) - */
        /* so we do so now.                                            */
        hdata->defered_release_flag = 0;
        kfree(actualFormat);
        MOD_DEC_USE_COUNT;
    }
    up(&hdata->sem);

    atomic_dec(&hdata->invocations);

    return orig_syscall_retval;
}
```

Annexe E

syscall_hijack_private.h

Cette annexe présente la spécification des fonctions et les ressources qui ne sont pas utilisées par des fichiers extérieurs.

```
//////////////////////////////////////
// type of info stored per syscall. //
//////////////////////////////////////
#define func_ptr unsigned long

struct syscall_hijack_data {
  func_ptr orig_syscall ;      /* for syscall restore.                  */
  struct semaphore sem ;       /* per-syscall semaphore.                */
  atomic_t invocations ;       /* how many current active invocations ? */
  int defered_release_flag ;   /* is there a pending (defered) release ? */
  char* format ;               /* info needed in NADF record */
} ;

struct hijacked_list_item{
  int index ;
  struct hijacked_list_item *next ;
} ;

//////////////////////////////////////
//templates des fonctions du module//
//////////////////////////////////////

static int auditmodule_ioctl(struct inode *, struct file *, unsigned int, unsigned long) ;
static int auditmodule_open(struct inode* node, struct file* the_file) ;
static int auditmodule_close(struct inode* node, struct file* the_file) ;
static ssize_t auditmodule_read(struct file *file, char *filebuffer, size_t length, loff_t *ppos) ;

//BUT : remplacer un appel système par un stub
//PRE : syscall_id est un index représentant la position dans la syscall_table de l'appel
```

```
//       système α remplacer                                                    30
//    format est un pointeur vers une chaîne de caractère définissant les informations
//       que nous désirons trouver dans le log α propos de cet appel système
//POST :-SI syscall_id < 0 | syscall_id >= NR_syscalls
//      ALORS -ENOSYS est retourné et rien n'est modifié
//    −SI un signal est réceptionné pendant l'obtention du sémaphore de la structure
//       syscall_hijack_data[syscall_id]
//      ALORS -ERESTARTSYS est retourné et rien n'est modifié
//    −SI le syscall a été précédemment hijacké
//      ALORS 1 est retourné et rien n'est modifié
//    -SI aucune stub_function n'existe pour le syscall                         40
//      ALORS −ENOENT est retourné et rien n'est modifié
//    -SI aucune des conditions précédentes n'est vrai
//      ALORS le syscall ayant l'index syscall_id est hijacké
//           ET syscall_hijack_data[syscall_id]->format = format
//           ET syscall_hijack_data[syscall_id]->defered_release_flag = 0
//           ET syscall_hijack_data[syscall_id]->orig_syscall pointe vers l'appel
//              système original
//           ET le "usage_count" du module est adapté afin d'empêcher un retrait
//              du module tant que le hijacking est actif
//           ET 0 est retourné                                                  50
int hijack_syscall(int syscall_id,char* format);

//BUT : Terminer le hijacking d'un appel système
//PRE : syscall_id est un index représentant la position dans la syscall_table de l'appel
//     système α replacer
//POST :-SI syscall_id < 0 | syscall_id >= NR_syscalls
//      ALORS -ENOSYS est retourné et rien n'est modifié
//    −SI un signal est réceptionné pendant l'obtention du sémaphore de la structure
//      syscall_hijack_data[syscall_id]
//      ALORS -ERESTARTSYS est retourné et rien n'est modifié                   60
//    −SI aucun hijacking n'est actif sur cet appel système
//      ALORS 0 est retourné et rien n'est modifié
//    −SI l'appel système était hijacké et que le stub n'est pas utilisé
//      ALORS syscall_hijack_data[syscall_id]−>orig_syscall est replacé dans la
//           syscall_table α l'index syscall_id
//           ET le 'usage count" du module est diminué de 1
//           ET l'espace mémoire alloué α syscall_hijack_data[syscall_id]->orig_syscall
//              est libérée
//    -SI l'appel système était hijacké et que le stub est utilisé
//      ALORS syscall_hijack_data[syscall_id]->orig_syscall est replacé dans la 70
//           syscall_table α l'index syscall_id
//           ET syscall_hijack_data[syscall_id]->defered_release_flag = 1
int release_syscall(int syscall_id);

int init_module(void);
void cleanup_module(void);
```

```
void init_hijack_syscalls(void) ;
void signal_auditd(void) ;

//BUT : ajouter record α la fin de la liste chaînée contenant tous les records    80
//    non encore transmis au user-mode
//PRE : -head pointe vers le premier record qui n'a pas encore été transmis au user-mode
//    -tail pointe vers le dernier record qui n'a pas encore été transmis au user-mode
//    -content est un pointeur vers une zone mémoire contenant un record NADF tel qu'il
//     doit être transmis au user-mode (mais sans padding α la fin du record)
//POST : tail pointe vers une struct record contenant record,
//     le contenu du champ de la struct record est content ET le padding nécessaire,
//     tail->next = NULL,
//     si head_pre == NULL alors head_post pointe vers la même struct contenant record
void addToListOfRecords(void* content) ;                                          90

//BUT : ajouter une information dans un buffer contenant un record NADF
//PRE : -base est un pointeur vers le buffer
//    -offset est la première position dans le buffer (nbr de bytes apd base)
//          ne contenant pas d'information utile
//    -id est l'id de l'information a ajouter (big endian notation)
//    -size représente la taille en bytes de l'information a ajouter (big endian notation)
//    -value est un pointeur vers un buffer contenant l'information α ajouter
//    -la taille du buffer value est >= size
//POST : -Si sizeOfBase >= offset+4+size+size%2                                    100
//     alors id, size et value sont ajouté au buffer base apd la position offset
//          (avec 1 byte de padding si size%2 != 0)
//          ET *sizeOfBase est inchangé
//          ET un pointeur vers le buffer base est retourné
//     -Si sizeOfBase < offset+4+size+size%2 ET il est possible d'allouer un nouvel espace
//        mémoire de la taille sizeOfBase+taille_initiale_de_base
//     alors un nouveau buffer (X) de taille sizeOfBase+taille_initiale_de_base contenant
//        la même information dans les offset premiers bytes est créer
//          ET *sizeOfBase est augmenté de taille_initiale_base
//          ET l'espace mémoire utilisé par base est libéré                      110
//          ET id, size et value sont ajouté au buffer X apd la position offset
//             (avec 1 byte de padding si size%2 != 0)
//          ET un pointeur vers le buffer X est retourné
//     -Si sizeOfBase < offset+4+size+size%2 ET il n'est pas possible d'allouer un nouvel
//        espace mémoire de la taille sizeOfBase+taille_initiale_de_base
//      alors un pointeur NULL est retourné
//           ET l'espace mémoire utilisé par base est libéré
void* addToNADFBuffer(void* base,unsigned long* sizeOfBase, unsigned long* offset
                 , unsigned short id, unsigned short size, void* value) ;
                                                                                  120

//////////////////////////
// information pour NADF //
```

```
//////////////////////////


#define PADDINGCHARACTER ' '
#define BASEBUFFERSIZE 1024
#define MAX_SIZE_OF_STRING 256
#define MAXCOMMAND 16                                                    130

////////////////////////////////////////////////
// structure d'un élément de la liste chaînée //
////////////////////////////////////////////////

struct record {
  void* content;
  struct record *next;
};
                                                                         140
////////////////////////////////////////////////
//compilation utile provenant de syscalltrack //
////////////////////////////////////////////////

#define sysctl_namelen_t int
#define versioned_string(x) (x)
#undef get_module_symbol
#define get_module_symbol(x,y) (y)
#ifndef MODULE_AUTHOR
#define MODULE_AUTHOR(x)                                                 150
#endif

#ifndef MODULE_DESCRIPTION
#define MODULE_DESCRIPTION(x)
#endif

#ifndef MODULE_LICENSE
#define MODULE_LICENSE(x)
#endif
                                                                         160
/* module owner field in struct file_operations */
#define MODULE_OWNER(x) owner : x,
```

Annexe F

syscall_hijack_public.h

Cette annexe présente la spécification des fonctions, des variables et des structures susceptibles d'être incluses dans d'autres fichiers.

```
////////////////////////////////
//Nom du fichier proc utilisé//
////////////////////////////////
#define AUDITDEV_NAME "masax"


//////////////////////////////////////
//définition des commandes pour ioctl//
//////////////////////////////////////
#define HIJACK 10
#define RELEASE 11
#define RELEASE_ALL 12

/////////////////////////////////////////////////////////
//structure utilisée pour la configuration du hijacking//
/////////////////////////////////////////////////////////
struct hijack_information{
  int syscall_index;
  char* format_string;
  struct hijack_information* next;
};
```

Annexe G

unixbench

Cette annexe présente les sources des exécutables utilisés lors des différents *benchmarks* issus du logiciel unixbench.

G.1 system call overhead

```
/*******************************************************************************
 *  The BYTE UNIX Benchmarks - Release 3
 *          Module : syscall.c  SID : 3.3 5/15/91 19 :30 :21
 *
 *******************************************************************************
 * Bug reports, patches, comments, suggestions should be sent to :
 *
 *      Ben Smith, Rick Grehan or Tom Yager at BYTE Magazine
 *      benbytepb.byte.com rick_gbytepb.byte.com   tyagerbytepb.byte.com
 *                                                                              10
 *******************************************************************************
 *  Modification Log :
 *  $Header : /etinfo/memor1/sep2003/clonfils_lievens/CVS-TFE/masax/book/annexes/syscall.c,v 1.1 2004/0
 *  August 29, 1990 - Modified timing routines
 *  October 22, 1997 - code cleanup to remove ANSI C compiler warnings
 *                     Andy Kahn <kahnzk3.dec.com>
 *
 ******************************************************************************/
/*
 *  syscall   - sit in a loop calling the system                                20
 *
 */
char SCCSid[] = "@(#) @(#)syscall.c :3.3 -- 5/15/91 19 :30 :21";

#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/stat.h>
#include "timeit.c"

unsigned long iter;                                                    30

void report()                                                          report
{
        fprintf(stderr,"%ld loops\n", iter);
        exit(0);
}

int main(argc, argv)                                                   main
int     argc;
char    *argv[];                                                       40
{
        int     duration;

        if (argc != 2) {
                printf("Usage : %s duration\n", argv[0]);
                exit(1);
        }

        duration = atoi(argv[1]);
                                                                       50
        iter = 0;
        wake_me(duration, report);

        while (1) {
                close(dup(0));
                getpid();
                getuid();
                umask(022);
                iter++;
        }                                                              60
        /* NOTREACHED */
}
```

G.2 spawn

```
/*****************************************************************************
 *  The BYTE UNIX Benchmarks - Release 3
 *          Module : spawn.c SID : 3.3 5/15/91 19 :30 :20
 *
 *****************************************************************************
```

```
 * Bug reports, patches, comments, suggestions should be sent to :
 *
 *      Ben Smith, Rick Grehan or Tom Yager at BYTE Magazine
 *      benbytepb.byte.com rick_gbytepb.byte.com   tyagerbytepb.byte.com
 *                                                                          10
 ******************************************************************************
 *  Modification Log :
 *  $Header : /etinfo/memor1/sep2003/clonfils_lievens/CVS-TFE/masax/book/annexes/spawn.c,v 1.1 2004/0
 *  August 29, 1990 - Modified timing routines (ty)                         routines
 *  October 22, 1997 - code cleanup to remove ANSI C compiler warnings
 *                     Andy Kahn <kahnzk3.dec.com>
 *
 ******************************************************************************/
char SCCSid[] = "@(#) @(#)spawn.c :3.3 -- 5/15/91 19 :30 :20";
/*                                                                          20
 *  Process creation
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include "timeit.c"

unsigned long iter;                                                         30

void report()
{
        fprintf(stderr,"%lu loops\n", iter);
        exit(0);
}

int main(argc, argv)                                                        main
int     argc;
char    *argv[];                                                            40
{
        int     slave, duration;
        int     status;

        if (argc != 2) {
                printf("Usage : %s duration \n", argv[0]);
                exit(1);
        }

        duration = atoi(argv[1]);                                           50

        iter = 0;
```

```
        wake_me(duration, report);

        while (1) {
                if ((slave = fork()) == 0) {
                        /* slave .. boring */
#if debug
                        printf("fork OK\n");
#endif                                                                  60
                        /* kill it right away */
                        exit(0);
                } else if (slave < 0) {
                        /* woops ... */
                        printf("Fork failed at iteration %lu\n", iter);
                        perror("Reason");
                        exit(2);
                } else
                        /* master */
                        wait(&status);                                  70
                if (status!= 0) {
                        printf("Bad wait status : 0x%x\n", status);
                        exit(2);
                }
                iter++;
#if debug
                printf("Child %d done.\n", slave);
#endif
                }
}                                                                       80
```

G.3 execl

```
/*****************************************************************************
 *  The BYTE UNIX Benchmarks - Release 3
 *          Module : execl.c  SID : 3.3 5/15/91 19 :30 :19
 *
 *****************************************************************************
 * Bug reports, patches, comments, suggestions should be sent to :
 *
 *      Ben Smith, Rick Grehan or Tom Yager
 *      benbytepb.byte.com rick_gbytepb.byte.com   tyagerbytepb.byte.com
 *                                                                          10
 *****************************************************************************
 *  Modification Log :
 *  $Header : /etinfo/memor1/sep2003/clonfils_lievens/CVS-TFE/masax/book/annexes/execl.c,v 1.1 2004/05/
```

```
 *  August 28, 1990 - Modified timing routines
 *  October 22, 1997 - code cleanup to remove ANSI C compiler warnings
 *                     Andy Kahn <kahnzk3.dec.com>
 *
 ***************************************************************************/
/*
 *  Execing                                                                  20
 *
 */
char SCCSid[] = "@(#) @(#)execl.c :3.3 -- 5/15/91 19 :30 :19";

#include <stdio.h>
#include <sys/types.h>

char    bss[8*1024];    /* something worthwhile */

#define main dummy                                                           30

#include "big.c"     /* some real code */

#undef main

/* added by BYTE */
char *getenv();


int main(argc, argv)  /* the real program */                                 40 main
int     argc;
char    *argv[];
{
        unsigned long iter = 0;
        char *ptr;
        char *fullpath;
        int     duration;
        char    count_str[6], start_str[12], path_str[81], *dur_str;
        time_t  start_time, this_time;
                                                                             50
#ifdef DEBUG
        int count;
        for(count = 0; count < argc; ++ count)
                printf("%s ",argv[count]);
                printf("\n");
#endif
        if (argc < 2)
                {
                printf("Usage : %s duration\n", argv[0]);
                exit(1);                                                     60
```

```
        }


    duration = atoi(argv[1]);
    if (duration > 0)
        /* the first invocation */
        {
        dur_str = argv[1];
        if((ptr = getenv("BINDIR"))!= NULL)
            sprintf(path_str,"%s/execl",ptr);                    70
        fullpath=path_str;
        time(&start_time);
        }
    else  /* one of those execl'd invocations */
        {
        /* real duration follow the phoney null duration */
        duration = atoi(argv[2]);
        dur_str = argv[2];
        iter = (unsigned long)atoi(argv[3]); /* where are we now? */
        sscanf(argv[4], "%lu", &start_time);                     80
        fullpath = argv[0];
        }

    sprintf(count_str, "%lu", ++iter); /* increment the execl counter */
    sprintf(start_str, "%lu", start_time);
    time(&this_time);
    if (this_time - start_time >= duration) { /* time has run out */
        fprintf(stderr, "%lu loops\n", iter);
        exit(0);
        }                                                        90
    execl(fullpath, fullpath, "0", dur_str, count_str, start_str, 0);
    printf("Exec failed at iteration %lu\n", iter);
    perror("Reason");
    exit(1);
}
```

G.4 shell

G.4.1 looper.c

```
char SCCSid[] = "@(#) @(#)looper.c :1.4 -- 5/15/91 19 :30 :22";

#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
```

```
#include "timeit.c"

unsigned long iter;
char *cmd_argv[28];
int  cmd_argc;                                                          10

void report(void)                                                       report
{
        fprintf(stderr,"%lu loops\n", iter);
        exit(0);
}

int main(argc, argv)                                                    main
int     argc;
char    *argv[];                                                        20
{
int     slave, count, duration;
int     status;

if (argc < 2)
        {
        printf("Usage : %s duration command [args. .]\n", argv[0]);
        printf("  duration in seconds\n");
        exit(1);
        }                                                               30

if((duration = atoi(argv[1])) < 1)
        {
        printf("Usage : %s duration command [arg. .]\n", argv[0]);
        printf("  duration in seconds\n");
        exit(1);
        }

/* get command */
cmd_argc=argc-2;                                                        40
for( count=2;count < argc; ++count)
        cmd_argv[count-2]=argv[count];
#ifdef DEBUG
printf("<<%s>>",cmd_argv[0]);
for(count=1;count < cmd_argc; ++count)
        printf(" <%s>", cmd_argv[count]);
putchar('\n');
exit(0);
#endif

                                                                        50
iter = 0;
wake_me(duration, report);
```

```
while (1)
        {
        if ((slave = fork()) == 0)
                { /* execute command */
                execvp(cmd_argv[0],cmd_argv);
                exit(0);
                }                                                              60
        else if (slave < 0)
                {
                /* woops ... */
                printf("Fork failed at iteration %lu\n", iter);
                perror("Reason");
                exit(2);
                }
        else
                /* master */
                wait(&status);                                                 70
        if (status != 0)
                {
                printf("Bad wait status : 0x%x\n", status);
                exit(2);
                }
        iter++;
        }
}
```

G.4.2 tst.sh

```
#! /bin/sh
##############################################################################
#  The BYTE UNIX Benchmarks - Release 3
#          Module : tst.sh  SID : 3.4 5/15/91 19 :30 :24
#
##############################################################################
# Bug reports, patches, comments, suggestions should be sent to :
#
#       Ben Smith or Rick Grehan at BYTE Magazine
#       benbytepb.UUCP rick_gbytepb.UUCP                                       10
#
##############################################################################
#  Modification Log :
#
##############################################################################
ID="@(#)tst.sh :3.4 -- 5/15/91 19 :30 :24";
```

```
sort >sort.$$ <sort.src
od sort.$$ | sort -n +1 > od.$$
grep the sort.$$ | tee grep.$$ | wc > wc.$$
rm sort.$$ grep.$$ od.$$ wc.$$
```

G.4.3 multi.sh

```
#! /bin/sh
##############################################################################
#  The BYTE UNIX Benchmarks - Release 3
#          Module : multi.sh SID : 3.4 5/15/91 19 :30 :24
#
##############################################################################
# Bug reports, patches, comments, suggestions should be sent to :
#
#       Ben Smith or Rick Grehan at BYTE Magazine
#       benbytepb.UUCP rick_gbytepb.UUCP
#
##############################################################################
#  Modification Log :
#
##############################################################################
ID="@(#)multi.sh :3.4 -- 5/15/91 19 :30 :24";
instance=1
while [ $instance -le $1 ]; do
        /bin/sh $BINDIR/tst.sh &
        instance=`expr $instance + 1`
done
wait
```